

**X . systems . press**

X.systems.press ist eine praxisorientierte  
Reihe zur Entwicklung und Administration von  
Betriebssystemen, Netzwerken und Datenbanken.

Joachim Schröder · Tilo Gockel · Rüdiger Dillmann

# Embedded Linux

Das Praxisbuch

Dipl.-Ing. Joachim Schröder  
Universität Karlsruhe  
Informatikfakultät  
Lehrstuhl IAIM Prof. Dr.-Ing. R. Dillmann  
Kaiserstraße 12  
76128 Karlsruhe  
schroede@ira.uka.de

Prof. Dr.-Ing. Rüdiger Dillmann  
Universität Karlsruhe  
Informatikfakultät  
Lehrstuhl IAIM Prof. Dr.-Ing. R. Dillmann  
Kaiserstraße 12  
76128 Karlsruhe  
dillmann@ira.uka.de

Dr.-Ing. Tilo Gockel  
Universität Karlsruhe  
Informatikfakultät  
Lehrstuhl IAIM Prof. Dr.-Ing. R. Dillmann  
Kaiserstraße 12  
76128 Karlsruhe  
goeckel@ira.uka.de

ISBN 978-3-540-78619-1

e-ISBN 978-3-540-78620-7

DOI 10.1007/978-3-540-78620-7

Springer Dordrecht Heidelberg London New York

X.systems.press ISSN 1611-8618

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

© Springer-Verlag Berlin Heidelberg 2009

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funk-sendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwider-handlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk be-rechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

*Einbandentwurf:* KünkelLopka, Heidelberg

Gedruckt auf säurefreiem Papier

Springer ist Teil der Fachverlagsgruppe Springer Science+Business Media ([www.springer.de](http://www.springer.de))

---

## Vorwort

Der Lehrstuhl von Herrn Prof. Dillmann am Institut für Anthropomatik der Universität Karlsruhe (TH) beschäftigt sich mit vielfältigen Themen rund um das Forschungsfeld der Robotik. Zu den Schwerpunkten zählen mobile Service-roboter und intelligente Fahrzeuge, maschinelles Lernen und Robotik in der Medizin, um nur einige davon zu nennen.

Aus den laufenden Forschungsarbeiten am Institut ergaben sich in den letzten Jahren zahlreiche Fragestellungen bei der Umsetzung spezieller Aufgaben auf Linux-Systemen. Die Themen betrafen die Auswahl geeigneter Hardware-Plattformen, die Komponentenanbindung, Netzwerkkommunikation, Software-Entwicklung für verschiedene Zielsysteme, Erstellung grafischer Benutzeroberflächen, Multithreading, Echtzeitfähigkeit und Smart-Camera-Technologie.

Die Antworten auf diese Fragen wurden zunächst in losen Blattsammlungen festgehalten, teilweise in elektronischer Form dokumentiert oder auch nur mündlich weitergegeben. Das vorliegende Buch hat zum Ziel, das Wissen zu strukturieren und zu bündeln. Die Inhalte wurden hierzu systematisch aufgearbeitet, um Grundlagenwissen ergänzt und durch viele Beispielapplikationen praxisgerecht veranschaulicht.

*Embedded Linux* ist mittlerweile das vierte Buch der Praxisbuchreihe, die nach dem Start mit dem Buch *Embedded Robotics* im Jahre 2005 mit dem *Praxisbuch Computer Vision*, und dessen englischsprachigem Pendant *Computer Vision – Principles and Practice*, weitergeführt wurde. Die zu diesen Themen vorliegenden Fachbücher sind oftmals als theoretische Lehrwerke konzipiert, entsprechend trocken zu lesen und wenig hilfreich in der Praxis. Die Praxisbücher sollen die Theorie ähnlich tiefgehend vermitteln, darüber hinaus aber die Inhalte um viele Versuche und Implementierungen ergänzen, die nicht nur im Laborumfeld, sondern auch im produktiven Umfeld standhalten.

Das entstandene Buch richtet sich an Studenten der Informatik und der Ingenieurwissenschaften, an Berufsanfänger, Praktiker und generell an alle Interessierten.

Die vorgestellten Bibliotheken und Applikationen werden bei Erscheinen des Buches online frei verfügbar sein. Der Download kann über den Link auf der Website des Springer-Verlages oder über <http://www.praxisbuch.net> erfolgen.

### *Danksagung*

Die Entstehung des vorliegenden Buches ist auch vielen anderen Menschen zu verdanken, die bei den Implementierungen mitgewirkt, die Ergebnisse korrigiert und viel Geduld mit den Autoren bewiesen haben. Es sind dies: Damira Mambetova, Ulla Scheich, Susanne und Detlef Schröder, Steven Wieland, Tobias Gindele, Manfred Kröhnert, Moritz Hassert, Alexander Bierbaum, Pedram Azad, Alexander Kasper, Peter Steinhaus und Andreas Böttinger. Besonders zu nennen sind an dieser Stelle auch Daniel Jagszent, der sein fundiertes Wissen als Gastautor in Kapitel 12 eingebracht hat und Stephan Riedel, der die Quelltext-Beispiele auf den verschiedenen Plattformen getestet hat.

Weiterhin möchten wir Herrn Prof. Rüdiger Dillmann für die Unterstützung und für das Vertrauen, das er in uns wissenschaftliche Mitarbeiter setzt, danken. Durch die hiermit verbundene Freiheit ist die Buchreihe der Praxisbücher erst möglich geworden. Abschließend danken wir Frau Dorothea Glaunsinger und Herrn Hermann Engesser vom Springer-Verlag für die Motivation, Geduld und die reibungslose und professionelle Zusammenarbeit.

Wir wünschen Ihnen viel Freude mit dem vorliegenden Buch und hoffen, Sie für den interessanten und zukunftssträchtigen Bereich der Embedded Systems unter Linux begeistern zu können.

Karlsruhe,  
den 5. Januar 2009

*Das Autorenteam*

### Hinweis

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Die erwähnten Soft- und Hardware-Bezeichnungen können auch dann eingetragene Warenzeichen sein, wenn darauf nicht gesondert hingewiesen wird. Sie sind Eigentum der jeweiligen Warenzeicheninhaber und unterliegen gesetzlichen Bestimmungen. Verwendet werden u. a. folgende geschützte Bezeichnungen: iPhone, Texas Instruments, Code Composer, Vision Components, Sony, KEIL,  $\mu$ Vision2.

---

# Inhaltsverzeichnis

---

## Teil I Grundlagen und Plattformen

---

<b>1</b>	<b>Grundlagen</b>	17
1.1	Einführung	17
1.2	Architekturen, Plattformen und Geschichtliches	18
1.3	Eigenschaften eingebetteter Systeme	21
1.3.1	Formfaktor	21
1.3.2	Mechanik, Kühlung, Robustheit	21
1.3.3	Speichermedien	22
1.3.4	Schnittstellen	23
1.3.5	Stromversorgung	24
1.3.6	Chipsätze	24
1.3.7	Watchdog	24
1.3.8	Echtzeitfähigkeit	25
1.4	Betriebssysteme	27
1.4.1	Allgemeine Anforderungen	27
1.4.2	Prozess-Scheduling	29
1.4.3	Systembeispiele	32
1.5	Software-Entwicklung	33
1.6	Aufbau und Gebrauch des Buches	38
<b>2</b>	<b>Hardware-Plattformen</b>	41
2.1	Einführung	41
2.2	Network-Attached-Storage NSLU2	42
2.3	WLAN-Router WL-500gP	44
2.4	MicroClient Jr. und Sr.	47
2.5	OpenRISC Alekto	51
2.6	Mini-ITX-Mainboard D945GCLF2 mit Dual-Core Atom CPU	52
2.7	Pegelanpassung für die RS-232-Schnittstelle	55

<b>3</b>	<b>OpenWrt auf dem WLAN-Router WL-500g Premium</b>	57
3.1	Einführung	57
3.2	Einrichtung des OpenWrt-Build-Systems	58
3.2.1	Aufspielen des Flash-Images	61
3.2.2	Der erste Einlog-Vorgang	62
3.3	Schnelleres Einloggen mit SSH-Keys	64
3.4	Software-Entwicklung für OpenWrt	65
3.5	Erstellung eigener OpenWrt-Module	67
3.6	IO-Warrior-Erweiterung und Kernelmodule unter OpenWrt	71
<b>4</b>	<b>Debian auf dem NAS-Gerät NSLU2</b>	75
4.1	Einführung	75
4.2	Debian-Installation	76
4.3	Erste Schritte	79
4.4	Software-Entwicklung für die NSLU2	80
4.5	NSLU2 als Druckerserver	81
4.6	Weiterführende Hinweise	84
4.6.1	Erstellung eines vollständigen NSLU2-Backups	84
4.6.2	Einstellung der Taster-Funktion	84
4.6.3	Probleme beim Booten	84
<b>5</b>	<b>Debian auf dem Embedded-PC OpenRISC-Alekto</b>	87
5.1	Einführung	87
5.2	Angepasste Debian-Installation	88
5.3	Erste Schritte	89
5.4	Software-Entwicklung	91
5.5	Zugriff auf die Alekto-Hardware	93
5.5.1	Anwendung der /proc-Erweiterungen in der Konsole	93
5.5.2	Zugriff über ioctl()-Befehle	94
5.6	Watchdog-Timer	96
5.7	Erstellung eines eigenen Alekto-Kernels	97
5.8	Vollständige Debian-Installation	98
<b>6</b>	<b>Puppy Linux auf dem Embedded-PC MicroClient Jr./Sr.</b>	101
6.1	Einführung	101
6.2	Puppy-Installation	101
6.3	Paket-Management unter Puppy	103
6.4	Software-Entwicklung unter Puppy	105

---

## Teil II Anwendungen

---

<b>7</b>	<b>Legacy-Schnittstellen und digitale IOs</b>	111
7.1	Einführung	111
7.2	RS-232	112



7.2.1	Grundlagen der RS-232-Schnittstelle .....	112
7.2.2	Ansteuerung und Programmierung .....	116
7.2.3	Ansteuerung einer seriellen Relaiskarte .....	121
7.3	Centronics und IEEE 1284 .....	124
7.4	General Purpose Input/Output (GPIO) .....	127
7.5	Schnittstellenerweiterung über IO-Warrior .....	129
7.5.1	IO-Warrior-Bausteine .....	129
7.5.2	Installation der IO-Warrior-Treiber unter Debian .....	130
<b>8</b>	<b>Der Inter-IC-Bus .....</b>	<b>133</b>
8.1	Einführung .....	133
8.2	I <sup>2</sup> C-Datenübertragung .....	136
8.2.1	Konzept .....	136
8.2.2	Steuersignale .....	136
8.2.3	Clock Stretching .....	137
8.2.4	Multi-Master-Betrieb .....	138
8.2.5	Adressierung .....	138
8.2.6	I <sup>2</sup> C-Buserweiterungen .....	141
8.3	I <sup>2</sup> C-Anbindung .....	144
8.3.1	I <sup>2</sup> C-Steckverbindung .....	146
8.3.2	Verwendung des I <sup>2</sup> C-Busses bei NSLU2 und Alekto ....	147
8.3.3	I <sup>2</sup> C-Busanbindung über einen IO-Warrior-Baustein ....	149
8.3.4	Die IO-Warrior-I <sup>2</sup> C-Bibliothek .....	150
8.4	Alternative serielle Bussysteme .....	152
8.4.1	Controller Area Network (CAN) .....	153
8.4.2	Local Interconnect Network (LIN) .....	154
8.4.3	1-Wire-Bus .....	155
8.4.4	Serial Peripheral Interface (SPI) .....	156
8.4.5	Universal Serial Bus (USB) .....	156
<b>9</b>	<b>Inter-IC-Bus-Komponenten .....</b>	<b>161</b>
9.1	Einführung .....	161
9.2	Die I <sup>2</sup> C-Bibliothek .....	163
9.2.1	Die Klasse IICBus .....	163
9.2.2	Die Klasse IICBase .....	165
9.3	Tastatur- und LC-Display-Ansteuerung mit PCF8574 .....	167
9.3.1	Philips 8-Bit-I/O-Erweiterung PCF8574 .....	167
9.3.2	I <sup>2</sup> C-Tastaturmodul .....	169
9.3.3	Die Klasse IICKeyboard .....	169
9.3.4	I <sup>2</sup> C-LC-Display .....	170
9.3.5	LC-Display-Treiberbaustein HD44780 .....	171
9.3.6	Die Klasse IICDisplay .....	173
9.3.7	Die Klasse IICIOExpander .....	176
9.4	Temperaturmessung mit DS1631 .....	177
9.4.1	Dallas DS1631 .....	177

9.4.2	Die Klasse IICTempSensor .....	178
9.5	A/D- und D/A-Wandler .....	179
9.5.1	Philips PCF8591 .....	179
9.5.2	Die Klasse IICADConverter .....	180
9.6	TMC222-Schrittmotorsteuerung .....	184
9.6.1	Trinamic TMC222 .....	184
9.6.2	Conrad C-Control I <sup>2</sup> C-Bus-Stepper-Driver .....	185
9.6.3	Die Klasse IICStepper .....	187
9.6.4	Programmierung des TMC222-OTP-Speichers .....	189
9.7	Chipkarten-Ansteuerung .....	190
9.7.1	EEPROM-Chipkarte AT24Cxx .....	191
9.7.2	Die Klasse IICChipcard .....	192
9.7.3	AES-Verschlüsselung .....	194
9.7.4	Die Klasse AES .....	197
9.8	I <sup>2</sup> C-Bus-Erweiterung über Multiplexer .....	199
9.8.1	Philips PCA9548 I <sup>2</sup> C-Multiplexer .....	199
9.8.2	Die Klasse IICMultiplexer .....	200
<b>10</b>	<b>USB-Komponenten .....</b>	<b>203</b>
10.1	Einführung .....	203
10.2	USB-Audioanbindung: MP3-Player und Sprachausgabe .....	204
10.3	USB-WLAN-Adapter .....	206
10.3.1	Grundlagen .....	206
10.3.2	Netgear MA111 unter Puppy .....	207
10.3.3	Alternative: WLAN-Anbindung über Access Point .....	209
10.4	USB-Bluetooth-Erweiterung .....	210
10.4.1	Grundlagen .....	210
10.4.2	Die Werkzeuge <i>Bluez-Utills</i> .....	211
10.4.3	Datentransfer mit ObexFTP .....	216
10.4.4	Serielle Bluetooth-Kommunikation und AT-Befehle .....	217
10.4.5	Das Mobiltelefon als Fernbedienung .....	219
10.5	USB-GPS-Module .....	222
10.5.1	Der GPS-Daemon GPSD .....	223
10.5.2	GPS in der Anwendung .....	224
10.5.3	Die Klasse GPSReceiver .....	225
10.6	USB-Speichererweiterung .....	226
10.6.1	Partitionierung und Einbindung eines USB-Sticks .....	226
10.6.2	Auslagerung des Home-Verzeichnisses auf einen USB-Stick .....	228
<b>11</b>	<b>Gerätetreiber und Kernelmodule .....</b>	<b>231</b>
11.1	Einführung .....	231
11.2	Grundlagen .....	232
11.2.1	Systemarchitektur .....	232
11.2.2	Der Kernel .....	234

11.3	Programmierung von Kernelmodulen .....	237
11.3.1	Aufbau von Kernelmodulen .....	237
11.3.2	Übersetzung von Kernelmodulen .....	239
11.3.3	Test und Debugging .....	239
11.3.4	Übergabe von Kommandozeilenparametern .....	242
11.4	Zeichenorientierte Gerätetreiber .....	243
11.4.1	Major-, Minor- und Geräteummern .....	243
11.4.2	Modul-Registrierung .....	245
11.4.3	Gerätetreiber-Registrierung nach alter Schule .....	248
11.5	Implementierung von Dateioperationen .....	249
11.5.1	Die Struktur <code>file_operations</code> .....	249
11.5.2	Kopieren von Daten zwischen Kernel- und User-Space ..	250
11.5.3	Die <code>ioctl()</code> -Schnittstelle .....	254
11.5.4	Verwendung von Gerätetreibern in der Anwendung ....	255
11.6	Hardware-Zugriff .....	257
11.6.1	Zugriff über IO-Ports und IO-Speicher .....	257
11.6.2	Zugriff über das Dateisystem .....	260
<b>12</b>	<b>Multithreading</b> .....	<b>263</b>
12.1	Einführung .....	263
12.2	Grundlagen .....	264
12.3	Posix-Schnittstelle .....	269
12.3.1	Thread-Funktionen .....	270
12.3.2	Mutex-Funktionen .....	271
12.3.3	Funktionen für Zustandsvariablen .....	272
12.3.4	Beispiel .....	273
12.4	C++-Schnittstelle .....	275
12.4.1	Die Klasse Thread .....	275
12.4.2	Die Klasse Mutex .....	277
12.4.3	Die Klasse WaitCondition .....	279
12.4.4	Die Klasse PeriodicThread .....	282
12.5	Anwendungsbeispiel: Servo-Ansteuerung .....	284
12.5.1	Servo-Anbindung an einen PC .....	285
12.5.2	Software-Entwurf zum Beispiel .....	286
12.5.3	Linux und Echtzeitfähigkeit .....	288
12.5.4	Zeitmessung .....	290
<b>13</b>	<b>Netzwerkkommunikation</b> .....	<b>295</b>
13.1	Einführung .....	295
13.2	Datenübertragung via UDP .....	297
13.2.1	Grundlagen zu Sockets .....	297
13.2.2	Berkeley Sockets .....	300
13.2.3	Verwendung der Berkeley Socket API .....	307
13.2.4	Socket-Debugging mit NetCat .....	310
13.2.5	Host Byte Order und Network Byte Order .....	310

13.2.6	PracticalSockets .....	312
13.2.7	Definition eigener Protokolle auf Anwendungsschicht ...	313
13.2.8	Verwendung der PracticalSockets .....	318
13.3	Kommunikation mit einer Qt-Anwendung .....	320
13.3.1	Client-Server-Kommunikation mit Qt4 .....	321
13.3.2	Remote-Schrittmotorsteuerung mit grafischer Benutzeroberfläche .....	327
13.4	Interaktion mit einem Webserver via CGI .....	333
13.4.1	Messdatenanzeige .....	336
13.4.2	Gezielte Anfragen mit JavaScript .....	338
<b>14</b>	<b>Video for Linux</b> .....	341
14.1	Einführung .....	341
14.2	Treiberinstallation und Inbetriebnahme .....	341
14.3	Bildeinzug unter Linux per V4L .....	345
14.4	Treiberkapselung für die IVT-Bibliothek .....	352
<b>15</b>	<b>Intelligente Kamera</b> .....	355
15.1	Einführung .....	355
15.2	Sicherheitssystem mit Bewegungserkennung .....	355
15.3	Weiterführende Informationen .....	358
15.3.1	Kommentare zum Hardware-Aufbau .....	358
15.3.2	Triggerung und IEEE 1394-Übertragung .....	360
15.3.3	Weitere Anwendungen .....	362
<b>16</b>	<b>Ausblick</b> .....	365
16.1	Communities, Projekte, Trends .....	365
16.2	Schlusswort und Kontaktdaten .....	369

---

## Teil III Anhang

---

<b>A</b>	<b>Kurzreferenzen</b> .....	373
A.1	Einführung .....	373
A.2	Die Linux-Konsole .....	373
A.2.1	Basisbefehlsschatz .....	373
A.2.2	Editoren .....	377
A.3	Netzwerkeinstellungen und SSH .....	380
A.3.1	Netzwerkeinstellungen .....	380
A.3.2	Secure Shell .....	382
A.4	Weitere Werkzeuge und Dienste .....	384
A.4.1	Paketverwaltung APT .....	384
A.4.2	Umgebungsvariablen .....	386
A.4.3	Erstellung von Gerätedateien mit <b>mknod</b> .....	387
A.4.4	Zugriffsrechte .....	388

A.4.5	Root-Rechte mit <code>sudo</code> .....	390
A.4.6	Cronjob-Verwaltung mit <code>crontab</code> .....	391
A.5	Diagnose- und Failsafe-Modi .....	393
A.5.1	Asus WL500g Premium .....	393
A.5.2	Linksys WRT54G .....	393
A.5.3	Linksys NSLU2 .....	394
<b>B</b>	<b>Alternative Hardware-Plattformen</b> .....	395
B.1	Einführung .....	395
B.2	Router .....	395
B.3	Network Attached Storage .....	395
B.4	Industrielle Kompaktsysteme .....	396
B.5	Einplatinencomputer .....	396
B.6	Sonderlösungen .....	396
<b>C</b>	<b>Die IVT-Bibliothek</b> .....	399
C.1	Einführung .....	399
C.2	Architektur .....	400
C.2.1	Die Klasse CByteImage .....	400
C.2.2	Anbindung von grafischen Benutzeroberflächen .....	401
C.2.3	Anbindung von Bildquellen .....	402
C.2.4	Anbindung der OpenCV .....	403
C.2.5	Anbindung von OpenGL über Qt .....	404
C.3	Beispielapplikationen .....	405
C.3.1	Verwendung der Basisfunktionalität .....	405
C.3.2	Verwendung einer grafischen Benutzeroberfläche .....	405
C.3.3	Verwendung eines Kameramoduls .....	405
C.3.4	Verwendung der OpenCV .....	406
C.3.5	Verwendung der OpenGL-Schnittstelle .....	406
C.4	Übersicht zu weiterer Funktionalität der IVT .....	407
C.5	Installation .....	408
C.5.1	OpenCV .....	409
C.5.2	Qt .....	409
C.5.3	Firewire und libdc1394/libraw1394 .....	410
C.5.4	IVT .....	411
<b>D</b>	<b>Die Qt-Bibliothek</b> .....	417
D.1	Einführung .....	417
D.1.1	Installation und Grundlagen .....	417
D.1.2	Signals und Slots .....	420
D.1.3	Ein universelles Qt-Makefile .....	424
D.2	Oberflächenerstellung mit Qt Designer .....	425
D.2.1	Installation und Grundlagen .....	425
D.2.2	Verwendung der Qt Designer Plugins .....	428
D.2.3	Erstellung der Qt Designer Plugins .....	430

**E    Bezugsquellen** ..... 435

**F    Verzeichnisbaum** ..... 439

**Literaturverzeichnis** ..... 441

**Sachverzeichnis** ..... 445

# Grundlagen

## 1.1 Einführung

Eingebettete Systeme oder auch englisch, *Embedded Systems*, begegnen uns mittlerweile überall im Alltag. Im Handy, in der Waschmaschine, Klimaanlage, Digitalkamera, im Auto, in der DSL- und Hifi-Anlage, in der Spielekonsole. Eine Beschäftigung mit diesem Thema ist entsprechend lohnenswert und interessant. Aber wie lautet denn die genaue Definition der mittlerweile relativ unscharf gebrauchten Bezeichnung? Zum Begriff der eingebetteten Systeme existiert keine exakte DIN- oder ISO-Definition. Im Sprachgebrauch wird der Begriff wie folgt verwendet:

*Der Begriff des eingebetteten Systems bezeichnet einen Rechner (Mikrocontroller, Prozessor, DSP<sup>1</sup>, SPS<sup>2</sup>), welcher dezentral in einem technischen Gerät, in einer technischen Anlage, allgemein in einem technischen Kontext eingebunden (eingebettet) ist. Dieser Rechner hat die Aufgabe, das einbettende System zu steuern, zu regeln, zu überwachen oder auch Benutzerinteraktion zu ermöglichen und ist speziell für die vorliegende Aufgabe angepasst.*

Dieser Versuch einer Definition umschließt mehrere Eigenschaften, einige explizit ausformuliert, einige als selbstverständlich zwischen den Zeilen vorausgesetzt: Ein eingebettetes System ist vor Ort bzw. dezentral im Gerät, Fahrzeug oder in der Anlage untergebracht. Es ist in Hard- und Software auf die jeweilige Aufgabe zugeschnitten und entsprechend auch so einfach und preiswert wie möglich gestaltet. Es benötigt Schnittstellen zum Prozess, zu anderen Geräten und zur Außenwelt – hierunter fällt auch die Benutzerschnittstelle zum Anwender. Wenn das eingebettete System eine schnelle Regelungsaufgabe bedient, so

---

<sup>1</sup> Digitaler Signalprozessor: Spezialprozessor für die Signalverarbeitung (Audio, Video; allgemein: schnelle mathematische Funktionen).

<sup>2</sup> Speicherprogrammierbare Steuerung: Steuerbaugruppe für die Automatisierungstechnik, programmierbar gem. der Norm EN 61131.

ist meist auch Echtzeitfähigkeit gefordert. Da das System weiterhin fest in den technischen Kontext eingebunden ist, ist es schlecht wartbar. Die Einheiten müssen entsprechend robust, wartungsarm und langlebig sein – eine Anforderung, die auch eine geringe Temperaturentwicklung, geringe Leistungsaufnahme und damit niedrige Taktraten nach sich zieht. Aus dem gleichen Grund sind auch mechanisch-bewegliche Komponenten wie Lüfter oder Festplatten in solchen Systemen eher selten zu finden.

Der letzte Nebensatz des Definitionsversuches beschreibt die dedizierte Anpassung an die gegebene Aufgabe. Entsprechend gelten auch im Sprachgebrauch beispielsweise Organizer oder PDAs nicht als eingebettete Systeme, obwohl sie viele der genannten Eigenschaften aufweisen. Diese Systeme sind bewusst nicht an eine spezielle Aufgabe angepasst, sondern universell gehalten und können eine Vielzahl von Aufgaben übernehmen.

Im weiteren Text werden die Charakteristika eingebetteter Systeme genauer untersucht und auch anhand von Beispielen belegt. Um einen guten Überblick zu bieten, wird dabei in diesem Grundlagenkapitel noch keine Einschränkung hinsichtlich linuxbasierter Systeme vorgenommen.

Das Kapitel schließt nach dieser allgemein gehaltenen Einführung mit einer Erläuterung zum Aufbau des vorliegenden Buches.

## 1.2 Architekturen, Plattformen und Geschichtliches

In Tabelle 1.1 wird die Entwicklung der Computertechnologie anhand der wichtigsten Meilensteine aufgezeigt. Hierbei wird die Mikrocontroller-Technologie (MC) speziell für eingebettete Systeme genutzt. In den frühen Anfängen wurde zwischen Mikroprozessoren und Mikrocontrollern noch nicht unterschieden; erst mit dem Aufkommen von Derivaten mit Daten- und Programmspeicher *on board* wurde die Bezeichnung Mikrocontroller geläufig.

Mittlerweile kennzeichnet dieser Begriff Bausteine, die generell höher integriert sind als Mikroprozessoren: Sie vereinen in monolithischer Bauweise den eigentlichen Mikroprozessor, Speicher, das Bussystem, A/D-D/A-Wandler und Schnittstellen zu seriellen Bussystemen (I<sup>2</sup>C, RS-232, RS-485, CAN ...). Mikrocontroller, die für spezielle Embedded-Anwendungen ausgelegt sind, besitzen darüber hinaus auch beispielsweise Hardware-Einheiten für die Pulsbreitenmodulation (Pulse Width Modulation, PWM), Quadratur-Decoder und Capture-Compare-Einheiten.

Erste Embedded-Anwendungen wurden direkt in der Maschinensprache des Controllers implementiert, später erwies sich die Hochsprache C als ausreichend maschinennah und erreichte eine weite Verbreitung. Mittlerweile werden komplexe Embedded-Anwendungen meist in C++ programmiert. Für be-



sonders zeitkritische oder maschinennahe Programmteile können Assembler-Anweisungen aber auch jetzt noch in C/C++ als sog. Inline-Assembler-Code eingebunden werden.

Neben der Entwicklung der höheren Programmiersprachen veränderten sich auch die Anforderungen an die Betriebssysteme. Erste Mikroprozessoren besaßen kein Betriebssystem, sondern nur einen sog. Bootstrap Loader, ein kleines Programm im ROM des Controllers, welches beim Start die hinterlegten Anwendungen lädt oder auch eine Verbindung zur Außenwelt zum Laden neuer Programme herstellt.

Die quasiparallele Bearbeitung mehrerer Aufgaben wird im einfachsten Fall direkt mit Interrupts realisiert. So können kurze Routinen vom Timer-Interrupt aufgerufen werden und auch externe Ereignisse einen (Hardware)-Interrupt auslösen. Häufig angeführte Beispiele sind Aufzugsteuerungen oder Geldautomaten. In beiden Beispielen muss der eingebettete Controller nicht nur die Regelung der Motoren (Seilwinde, Türmechanik, Transportmechanik) und das Auswerten der Sensoren im System (Drehgeber, Endschalter) abdecken, sondern auch eine Benutzerschnittstelle zur Außenwelt bedienen. Im einfachen Fall des Aufzuges ist dies ein Bedien-Panel mit Tasten und eine Siebensegmentanzeige. Im etwas komplexeren zweiten Beispiel werden im Regelfall eine numerische Tastatur und ein Monitor eingesetzt.

Unter Verwendung des Timer-Interrupts und der damit möglichen Zuteilung von Zeitscheiben zu Prozessen sind auch auf einfachen Mikrocontrollern Multi-Threading-Anwendungen möglich, die Programmierung wird allerdings rasch komplex und fehleranfällig. Mit der zusätzlichen Forderung der Möglichkeit einer Schnittstellenkapselung bzw. eines Treiberkonzeptes wurde entsprechend der Wunsch nach einem Betriebssystem laut. Weiterhin werden mittlerweile auch MP- und DSP-Boards in immer kleinerer Bauform hergestellt, und parallel steigt auch bei Embedded-Anwendungen der Ressourcenbedarf.

Entsprechend werden auch für diese Anwendungen seit längerer Zeit nicht nur Mikrocontroller, sondern auch leistungsstarke Prozessoren mit leistungsfähigen Betriebssystemen eingesetzt. Typische Vertreter sind die ARM7/ARM9-Prozessorfamilie, die TMS320-DSPs, die Atom-CPU von Intel, die Modellreihe VIA Nano von VIA Technologies und die Geode-Familie von AMD.

Anzumerken ist, dass fast alle genannten Mikrocontroller-Technologien und Entwicklungstechniken auch heute noch Anwendung finden. So haben moderne 32-bit-Controller die ersten 4- und 8-bit-Controller nicht gänzlich abgelöst, sondern nur ergänzt. Ein Beispiel hierzu: die veraltete 8051-Technologie wird auch aktuell noch von rund zwölf Herstellern angeboten, wobei auch modernste Derivate einhundertprozentig befehlskompatibel zu der ersten Generation sind.

Ein Grund hierfür ist der wachsende Preisdruck. Für jede Anwendung wird der preiswerteste Controller verwendet, der für die Aufgabe gerade noch in Frage kommt. Weiterhin ist der Markt der eingebetteten Systeme ein träger

Jahr	Entwicklungsschritt	Kenngrößen	Anmerkungen
1948	Herstellung erster Transistoren, William B. Shockley	Zuerst JFET	Nach dem Patent von Julius Edgar Lilienfeld von 1925
1955	Herstellung des ersten Computers mit Transistoren: der TRADIC, Bell Laboratories		TRansistorized Airborne DIgital Computer, entwickelt für die United States Air Force
1958	Erster integrierter Schaltkreis, Jack S. Kilby	monolithische Bauweise	Patent: „Solid Circuit made of Germanium“
1961	Apollo Guidance Computer (AGC), Charles Stark Draper / MIT	monolithische Bauweise	Steuerrechner für das Apollo-Raumfahrt-Programm
1961	Autonetics D-17 Guidance Computer	monolithische Bauweise	Steuerrechner für die Minuteman-Raketen; erste Mikroprozessor-Serienfertigung
1971	4004, Intel	4 bit	Erste integrierte Mikroprozessoren für Taschenrechner
1972	8008, Intel	8 bit	Erster 8-Bit-Prozessor
1974	TMS1000, Texas Instruments	4 bit	Erster Mikrocontroller
1974	8080, Intel	8 bit	MP
1974	6800, Motorola	8 bit	MP
1975	8048, Intel	8 bit	Erster MC mit RAM und ROM
1976	8748, Intel	8 bit	MC mit EPROM
1976	TMS9000, Texas Instruments	16 bit	MP
1977	Z80, Zilog	8 bit	MP
1978	6805, Motorola	8 bit	MC
1978	8086, Intel	16 bit	MP
1979	68000, Motorola	16 bit	MP
1979	2920, Intel	16 bit	Erster DSP
1981	8051, Intel	8 bit	MC
1983	TMS32010, Texas Instruments	32 bit	Damals schnellster DSP auf dem Markt; Grundsteinlegung der TMS320-DSP-Familie
1983	IMS T400, INMOS	32 bit	Erster Transputer für die parallele Datenverarbeitung
1984	68HC11, Motorola	8 bit	MC
1984	68020, Motorola	32 bit	MP
1984	Erster FPGA, Ross Freeman / Xilinx		Field-Programmable Gate Array (FPGA), frei programmierbarer Logikbaustein
1985	80386, Intel	32 bit	MP
1987	ARM2, Acorn Computers Ltd.	32 bit RISC	Aktueller Stand: ARM9
1989	PICmicro, Microchip Technology Inc.	8 bit RISC	Low-Cost-MC; mittlerweile auch 16 bit, 32 bit
1991	Erster FPAA, Edward Lee und Glenn Gulak / Universität Toronto		Field-Programmable Analog Array (FPAA), wie FPGA, zus. auch analoge Schaltkreise
1993	C166/167, Siemens	16 bit	MC
bis 2003	x86-Prozessorfamilie, Intel	32 bit	MP; Intel ist Marktführer; kleinere Hersteller: AMD, Cyrix
bis 2008	x86-Prozessorfamilie, Intel, AMD	32 bit	MP; Intel und AMD
2005	Cell-Prozessorfamilie, Sony, Toshiba, IBM (STI)	64 bit Multi-Core PowerPC	MP, Bsp.-Applikation: IBM Blade-PC; Sony Playstation 3
2008	Intel Atom, Intel	32 / 64 bit Low Power	Low-Power-Design; Einsatz in Mobile PCs und Embedded PCs
2008	VIA Nano, VIA Technologies	64 bit x86 / x86-64 Low Power	Low-Power-Design; höhere Leistung als Intel Atom; demnächst: Dual-Core-Variante

**Tabelle 1.1.** Entwicklung der Mikroprozessoren und Mikrocontroller über die Zeit (vgl. auch [Beierlein 04]).

Markt. Die Maschinenbauer und Automatisierungstechniker weichen nur ungern von einer bewährten Technik ab, da die Ansprüche an die Systeme hinsichtlich Sicherheit und Verfügbarkeit hier wesentlich höher sind als im PC- bzw. Consumer-Markt. Die Nutzung einer bewährten Technologie über einen Zeitraum von bis zu zehn Jahren ist keine Seltenheit, und so muss auch der Halbleiter-Hersteller die langfristige Verfügbarkeit sichern.<sup>3</sup>

## 1.3 Eigenschaften eingebetteter Systeme

Einige der Eigenschaften eingebetteter Systeme wurden bereits in der Einführung angesprochen. Diese Charakteristika werden nun weiter ausgeführt.

### 1.3.1 Formfaktor

Der Formfaktor eingebetteter Systeme ist im Regelfall ein anderer als bei Standard-PCs, da die Baugruppen kompakter aufgebaut sind. Die gängigen Formfaktoren sind in Abbildung 1.1 zusammengestellt.

Ergänzend seien angeführt: das Euro-Format ( $160 \times 100 \text{ mm}^2$ ), das halbe Euro-Format ( $80 \times 100 \text{ mm}^2$ ) und das VESA 75/100-Format<sup>4</sup> für Geräte, die auf die Aufnahme an der Rückseite eines Standard-VESA-Monitors passen. Es existiert darüber hinaus aber auch eine Vielzahl von herstellerspezifischen Sonderformaten.

### 1.3.2 Mechanik, Kühlung, Robustheit

Eingebettete Systeme besitzen im Regelfall keine mechanisch-beweglichen Komponenten. Die Prozessoren werden mit moderaten Taktraten betrieben und können entsprechend lüfterlos passiv gekühlt werden. Der Vorteil eines niedrig getakteten, lüfterlosen Systems liegt auf der Hand: Die Komponenten erwärmen sich weniger stark und haben daher eine längere Lebensdauer, das Gehäuse benötigt keine Lufteinlassöffnungen und kann daher staubdicht ausgeführt werden (vgl. auch Schutzarten: IP54, IP65 usw.).

Darüber hinaus werden eingebettete Systeme oft auch für einen größeren Temperaturbereich spezifiziert, da die Prozessoren beispielsweise im Kfz einer viel größeren Temperaturschwankung ausgesetzt werden als im Standard-PC im Büro. Je nach Einbauposition des Systems in der Fahrgastzelle kann hier ein Temperaturbereich von bis zu  $-50^\circ\text{C}$  bis  $+120^\circ\text{C}$  erforderlich sein.

<sup>3</sup> Beispiele: AMD garantiert fünf, Intel mittlerweile sieben Jahre Verfügbarkeit.

<sup>4</sup> Die Aufnahmebohrungen befinden sich an den Ecken eines Quadrates mit der Kantenlänge 75 bzw. 100 mm. Der Monitor besitzt passend hierzu an der Rückseite vier M4-Gewinde.

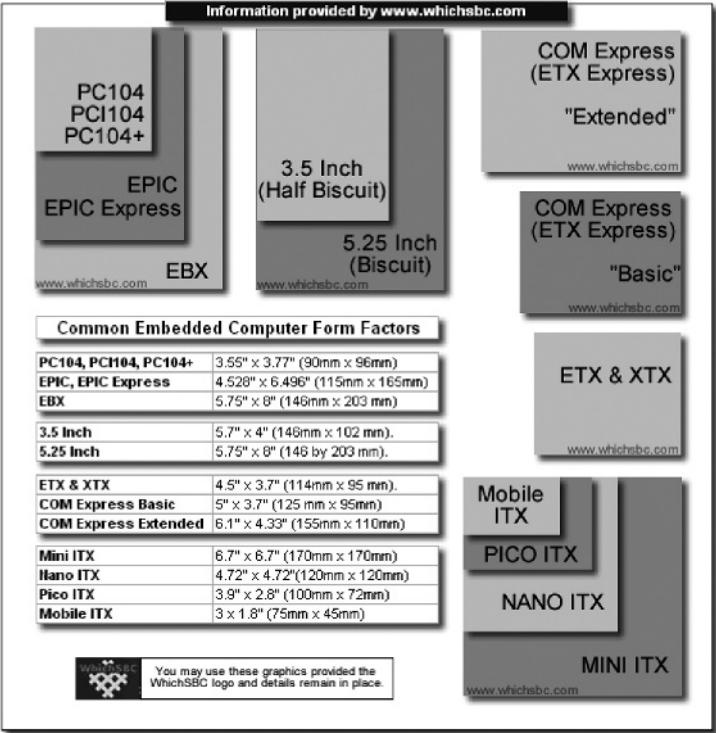


Abb. 1.1. Gängige Formfaktoren in der Industrie [WhichSBC 08].

1.3.3 Speichermedien

Als Festspeichermedien kommen selten Festplatten, meist Flash-Speicher im Controller oder auf dem Board oder auch in Form von SD- oder CF-Cards zum Einsatz [Wikipedia 08, Solid\_State\_Drive]. Der Grund hierfür ist die höhere Verfügbarkeit bzw. Robustheit dieser Medien. Weiterhin ist oft ein Verzicht auf mechanisch-bewegliche Komponenten gefordert, und auch die Einbaulage des Systems soll frei wählbar sein – mechanische Festplatten weisen hier oftmals Beschränkungen auf.

Beim Einsatz von Flash-Speichermedien ist zu beachten, dass die Anzahl der Schreibzugriffe minimiert werden sollte. Zwar verfügen moderne CF- und SD-Karten und USB-Sticks über internes *Block Journaling* und *Wear Leveling*<sup>5</sup>, sie können aber hinsichtlich der Langlebigkeit bei vielen Schreibzugriffen mit modernen Festplatten noch nicht gleichziehen. Bei Karten ohne eigenen Controller, z. B. vom Typ SmartMedia (SM), ist ohnehin größte Vorsicht geboten.

<sup>5</sup> Es handelt sich um Optimierungsverfahren, die die Schreibzugriffe pro Sektor minimieren.

Hier empfiehlt sich die Verwendung eines geeigneten Dateisystems wie beispielsweise jffs2<sup>6</sup>.

Weiterhin kann auch durch die Überdimensionierung des Speichermediums ein gewisses Maß an Sicherheit und Langlebigkeit erkaufte werden – der Faktor ist näherungsweise gleich ( $n$ -fach überdimensioniert bedeutet näherungsweise  $n$ -fache Anzahl möglicher Schreibzugriffe).

Eine weitergehende Untersuchung zur Langzeittauglichkeit von Flashspeichermedien ist zu finden unter [ct 08, Heft 23/06, S. 136]. Hier wurden im Rahmen eines Langzeittests auf einer Datei 16 000 000 Schreiboperationen durchgeführt, und die Datei war noch immer fehlerfrei lesbar. In diesem Test wurde nicht auf physikalische, sondern auf logische Sektoren geschrieben. Entsprechend ist hieraus eher auf die Leistungsfähigkeit der Wear-Leveling-Algorithmen, als auf die Haltbarkeit der Flash-Bausteine zu schließen.

Bei den in den Speichermedien eingesetzten Flash-Speicherbausteinen in NAND-Bauweise existieren zwei Bauformen: In den Standard-Consumer-Medien werden sog. Multi-Level-Cell-Chips (MLC-Chips) verwendet, für die die Hersteller um die 10 000 Schreibzyklen angeben. Bei den höherwertigen Medien werden Single-Level-Cell-Chips (SLC-Chips) verbaut, die rund dreimal schneller beschreibbar sind und auch eine höhere Lebensdauer von ca. 100 000 Schreibzyklen aufweisen (Quelle: der *Flash Memory Guide* von Kingston, [Kingston 08]). Nach der Information, in welchen Produkttypen nun SLC- bzw. MLC-Chips verbaut sind, muss der Anwender u. U. ein wenig suchen. Für Kingston-Medien findet sich die Information auf der letzten Seite des *Flash Memory Guide*.

### 1.3.4 Schnittstellen

Eingebettete Systeme besitzen im Regelfall zusätzliche Schnittstellen symmetrischer Art (RS-422), Busschnittstellen (Inter-IC-Bus (I<sup>2</sup>C)), Feldbusschnittstellen (RS-485; CAN und Profibus auf RS-485), digitale Ein- und Ausgänge (DIOs), analoge Ein- und Ausgänge und Schnittstellen zu bestimmten LC-Displays.

Auch auf Standard-PC-Boards lassen sich die meisten dieser Schnittstellen nachrüsten, und so kommen immer häufiger USB-IO-Wandler, USB-RS-485-Wandler oder ähnliche Baugruppen zum Einsatz.

Etwas komplexer wird der Sachverhalt, falls echtzeitfähige IOs gefordert sind. Hier versagen Umsetzer von USB nach I<sup>2</sup>C oder von Ethernet nach DIO, da die

---

<sup>6</sup> Journaling Flash File System, ein Dateisystem, welches speziell für Flash-Speichermedien ausgelegt ist und die Wear-Leveling-Funktionalität bereits enthält.

zwischen geschalteten seriellen Übertragungsstrecken zu große Latenzen aufweisen. Dies ist auch ein Grund, weswegen die parallele Schnittstelle gem. IEEE 1284 zumindest im Embedded-Bereich noch lange nicht ausgedient hat.

### 1.3.5 Stromversorgung

Die Stromversorgung eingebetteter Systeme ist im Regelfall einfach aufgebaut. Die Systeme kommen meist mit einer geringen Anzahl von Versorgungsspannungen aus (Bsp.: durchgängig 5 VDC) bzw. generieren weitere Spannungen *on board*, und weisen auch durch die kleineren Taktraten und das Fehlen mechanischer Komponenten einen geringen Stromverbrauch auf.

Ein weiterer Vorteil hierbei ist die Möglichkeit, einen mobilen Akku-Betrieb einfacher umsetzen zu können. Anwendungen finden sich beispielsweise im Kfz oder auch in einem fahrerlosen Transportfahrzeug (FTF) in der Fertigung.

### 1.3.6 Chipsätze

Die in eingebetteten Systemen verwendeten Chipsätze sind oft andere als bei Standard-PCs. Zum einen bieten die Chipsätze teilweise zusätzliche Schnittstellen oder Funktionalitäten<sup>7</sup>, zum anderen müssen sie auch sehr viel länger lieferbar sein, da Industriekunden eine Sicherheit von bis zu zehn Jahren für die Ersatzteilstellung fordern. Weiterhin sind die verwendeten Chipsätze und Prozessoren nicht auf Geschwindigkeit optimiert, sondern auf Robustheit, kleinen Platzbedarf und geringe Herstellungskosten.

Aktuell zeichnet sich hierbei ein interessanter neuer Trend ab: Die Low-Power-Embedded-Prozessoren und Chipsätze Atom (Fa. Intel) und Nano (Fa. VIA) haben sich auch als sehr geeignet erwiesen für den Einsatz in der jungen Produktfamilie der sog. Netbooks<sup>8</sup>.

Dieser neue Trend im Consumer-Markt hin zu stromsparenden, robusten und preiswerten PCs wird voraussichtlich auch weiterhin auf den Markt der Embedded-PCs einen großen Einfluss haben.

### 1.3.7 Watchdog

Eingebettete Systeme verfügen im Regelfall über einen sog. Watchdog. Es handelt sich hierbei um einen Zähler, der hardwareseitig ständig inkrementiert

<sup>7</sup> Bspw. sog. General Purpose Inputs/Outputs (GPIOs).

<sup>8</sup> Preiswerte, besonders baukleine Notebook-PCs ohne Laufwerke, teilweise auch ohne mechanische Festplatte.

wird und der beim Überlauf (im Fehlerfall) das System neu startet. Entsprechend muss der Zähler im fehlerfreien Betrieb vom laufenden Hauptprogramm oder auch vom Betriebssystem regelmäßig zurückgesetzt werden. Dieserart ist gewährleistet, dass das System, das u. U. schwer zugänglich in einer Anlage verbaut ist, im Fehlerfall selbstständig wieder anläuft.

Je nachdem, welcher Prozess den Zähler zurücksetzt, können blockierende Fehler im Betriebssystem und/oder Fehler im Anwenderprogramm erkannt werden. Zu weiteren Details vgl. Abschnitt 5.6 und die dort vorgestellte Watchdog-Implementierung.

### 1.3.8 Echtzeitfähigkeit

Der Begriff der Echtzeit bzw. Real-time wird mittlerweile nicht nur im Zusammenhang mit schnellen Regelungen, sondern auch in vielen anderen Bereichen wie z. B. im Multimedia-Bereich verwendet und wurde entsprechend in den letzten Jahren ein wenig verwaschen. An dieser Stelle sollen nun für den Gebrauch im vorliegenden Buch die Begriffe nochmals klar dargelegt werden.

Die DIN 44300, Teil 9 definiert den Begriff der Realzeitverarbeitung (Real-time Processing) wie folgt: *Eine Verarbeitungsart, bei der Programme zur Verarbeitung anfallender Daten ständig ablaufbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu bestimmten Zeitpunkten anfallen.*

Hier wird entsprechend gleichgesetzt: *Echtzeit = Rechtzeitigkeit*. Tatsächlich kann die Verarbeitung aber auch zu schnell erfolgen, bzw. es können Ergebnisse zu früh vorliegen. Je nach System sind die folgenden unterschiedlichen Zeitbedingungen einzuhalten (vgl. auch [Wörn 05]):

**Exakter Zeitpunkt:** Bei dieser zeitlichen Bedingung wird ein exakter Zeitpunkt  $t_0$  definiert, bei welchem die Aktion stattfinden muss. Wird der Zeitpunkt nicht eingehalten, so arbeitet das System nicht mehr innerhalb seiner Spezifikation. Ein Beispiel hierfür ist das Einfahren eines ICE-Personenzuges in den Bahnhof. Hier muss der Bremsvorgang exakt zum gegebenen Zeitpunkt  $t_0$  erfolgen, da sonst die Wagen nicht mehr innerhalb der ausgewiesenen Bahnsteigbereiche zum Halten kommen.

**Spätester Zeitpunkt:** Hier wird ein maximaler Zeitpunkt  $t_{max}$  angegeben, bis zu welchem ein Ereignis stattfinden muss. Ein typisches Beispiel ist das Halten einer Lore in einem fahrerlosen Transportsystem an einem gegebenen Haltepunkt (rote Ampel). Das Fahrzeug darf auf keinen Fall später halten und in den Kreuzungsbereich hineinfahren. Ein etwas verfrühter Halt ist aber vertretbar.

**Frühester Zeitpunkt:** Hierbei wird ein minimaler Zeitpunkt  $t_{min}$  angegeben, der einzuhalten ist. Ein klassisches Beispiel ist die Abhängigkeit von einem anderen Ereignis. So darf das Entladen einer Lore frühestens dann erfolgen, wenn diese die Entladestation erreicht hat, aber auch später.

**Zeitintervall:** Hier wird ein Zeitbereich zwischen  $t_{min}$  und  $t_{max}$  für eine mögliche Bearbeitung angegeben. Ein Beispiel hierfür ist wieder der Entladevorgang der Lore. Je nach Systemkonfiguration muss die Lore zu einer bestimmten Zeit (spätestens) den Entladeplatz verlassen haben, da bereits die nächste Lore ankommt.

**Absolute und relative Zeitbedingung:** Eine absolute Zeitbedingung ist in Form einer Uhrzeit angegeben. Ein Beispiel: Das Flugzeug muss frühestens, exakt oder spätestens um 15:00 Uhr starten. Eine relative Zeitbedingung ist in Form eines Zeitbereiches und in Abhängigkeit von einem anderen Ereignis angegeben. Ein Beispiel: Der Stellwert in einer Regelung muss frühestens, exakt oder spätestens 2s nach dem Vorliegen des Istwertes ausgegeben werden.

**Periodizität:** Periodische Ereignisse sind immer wiederkehrend. Ein typisches Beispiel ist eine Regelstrecke, bei welcher nach jedem Regelintervall die errechneten Stellwerte ausgegeben und neue Istwerte von den Sensoren eingelesen werden.

**Synchronität:** Bei einem synchronen System erfolgt die zeitliche Abarbeitung der Prozessschritte in einem festen Raster, gebunden an einen Master-Takt. Ein Beispiel hierfür ist ein System zur Achslagerregelung. Hier ist die Abtaststrategie des unterlagerten Drehzahlreglers zeitlich synchron an jene des Lagereglers gebunden bzw. ein ganzzahliges Vielfaches davon (Abbildung 1.2).

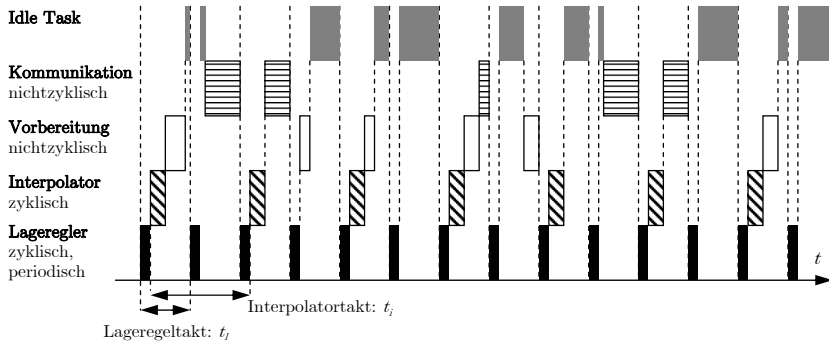
Abbildung 1.2 zeigt ein Beispiel für ein System mit mehreren parallelen Echtzeit-Tasks. Hier werden auch die Begriffe Periodizität und Synchronität rasch klar.

**Harte Echtzeit:** Die Anforderung der harten Echtzeit liegt dann vor, wenn bei Nichteinhalten der Zeitbedingung Schaden droht. Ein typisches Beispiel ist der besagte Transportroboter, der auf eine rote Ampel zufährt. Für die harte Echtzeitbedingung gilt: Wenn das System die Bedingung verletzt, so arbeitet es außerhalb seiner Spezifikation und muss in einen sicheren Zustand überführt werden (abgeschaltet werden).

**Feste Echtzeit:** Die Anforderung der festen Echtzeit liegt dann vor, wenn bei Nichteinhalten der Bedingung zwar kein Schaden droht, aber das Ergebnis der Operation wertlos wird.

**Weiche Echtzeit:** Die Bedingung der weichen Echtzeit liegt vor, wenn die vorgegebenen Zeiten eher als Richtgrößen, denn als feste Vorgaben zu sehen sind. Ein typisches Beispiel ist ein Multimedia-System, bei welchem bei der





**Abb. 1.2.** Eine Robotersteuerung als Beispiel für ein echtzeitfähiges, synchrones System.

Filmwiedergabe ein selten auftretendes Ruckeln noch problemlos toleriert werden kann.

Die verschiedenen Möglichkeiten der Realisierung dieser Bedingungen werden im nachfolgenden Abschnitt 1.4 weiter ausgeführt.

## 1.4 Betriebssysteme

### 1.4.1 Allgemeine Anforderungen

Wie bereits angesprochen, ist für ein einfaches eingebettetes System ein Betriebssystem nicht unbedingt erforderlich. Im einfachsten Fall der diskreten Regelschleife besteht das Programm aus einer einzigen Hauptschleife, welche ohne Unterbrechung (ohne Interrupts) ausgeführt wird. Am Ende eines Schleifendurchlaufes werden die berechneten Größen ausgegeben und von den Sensoren neue Istwerte eingelesen.

Derart einfache Systeme sind in der Praxis nur selten ausreichend. Meist ist neben der Regelung noch eine komplexe Kommunikationsroutine aufzurufen, es sind externe Interrupts von Achs-Endschaltern zu bearbeiten und Ausgaben weiterer Sensoren für Temperatur u. ä. zu behandeln. Oft kommt weiterhin noch eine Benutzerschnittstelle hinzu. Spätestens jetzt ist der Einsatz eines Betriebssystems sinnvoll; es organisiert und priorisiert den quasiparallelen Ablauf der einzelnen Tasks in Zeitscheiben, es bietet eine Kapselung für Schnittstellen, verwaltet den Arbeitsspeicher und regelt den Zugriff auf Festspeichermedien über ein Dateisystem.

Abweichend von den klassischen Betriebssystemen, wie sie bei Desktop-PCs zum Einsatz kommen, wird bei eingebetteten Systemen weiterhin oft die Datenverarbeitung zu harten Echtzeitbedingungen gefordert. Hierzu wiederum eine Definition:

*Ein Echtzeitbetriebssystem ist ein Betriebssystem, das die Konstruktion eines Echtzeit-Systems erlaubt.* [Marwedel 07]

Aus diesen Anforderungen an ein Echtzeitbetriebssystem erwachsen auch bestimmte Anforderungen an die Interrupt-Behandlung und an das Scheduling-Verfahren (vgl. Abschnitt 1.4.2). In Tabelle 1.2 sind die Eigenschaften von konventionellen Betriebssystemen und Echtzeitbetriebssystemen einander gegenübergestellt.

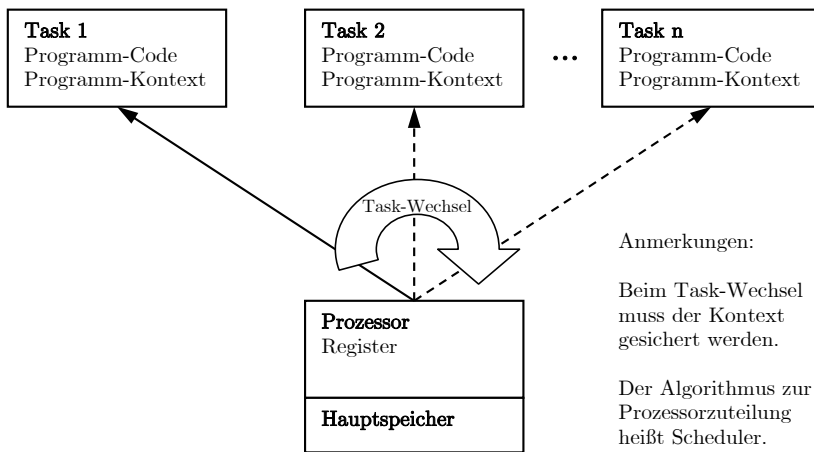
Konventionelles Betriebssystem	Echtzeit-Betriebssystem (RTOS)
Logisch korrekte Ergebnisse	Logisch und zeitlich korrekte Ergebnisse
Keine Garantie für die maximale Antwortzeit, Zeitüberschreitungen sind tolerierbar	Garantierte Antwortzeit (Latenz); bei harten Echtzeitanforderungen sind Zeitüberschreitungen fatal
Effizientes, aber nicht einfach vorhersagbares Scheduling	Vorhersagbares und kontrollierbares Scheduling
Optimiert auf maximalen Datendurchsatz	Optimiert auf minimale Antwortzeiten, typisch um die 20 $\mu$ s
Optimiert auf den durchschnittlichen Lastfall	Optimiert auf den maximalen Lastfall (Worst Case)
Langsamer Scheduler, geringe zeitliche Auflösung und Genauigkeit (Standard-Linux: 10 ms)	Schneller Scheduler, hohe zeitliche Auflösung und Genauigkeit, typisch: 20–200 $\mu$ s
Standard-Scheduling-Verfahren: Time-Sharing, First Come First Served, Round-Robin, Round-Robin mit Prioritätsklassen	Deterministische Scheduling-Verfahren wie Earliest Deadline First; kurze und deterministische Zeiten für Taskwechsel
Nur Systemprozesse laufen im Kernelmode	Systemprozesse und zeitkritische Anwenderprozesse laufen im Kernelmode
Periodischer Timer-Interrupt	Nicht zwingend periodischer, aber hochaufgelöster Timer-Interrupt (Stichwort: One-Shot-Timer)
Teilweise langes Sperren (Maskieren) der Interrupts	Schnelle Interrupt-Behandlung

**Tabelle 1.2.** Gegenüberstellung der Eigenschaften der Betriebssysteme.

### 1.4.2 Prozess-Scheduling

Der sog. Scheduler ist jener Teil des Betriebssystems, der den Prozessen bzw. Tasks den Prozessor bzw. die Rechenzeit zuteilt.<sup>9</sup> Für diese Zuteilung sind mehrere Algorithmen gebräuchlich, wobei die Auswahl von der jeweiligen Anwendung abhängt (Abbildung 1.3).

So steht bei einfachen Ablaufsteuerungen ein hoher Datendurchsatz im Vordergrund, wohingegen von Systemen mit Benutzerinteraktion eine schnelle Antwortzeit gefordert wird. Echtzeit-Systeme wiederum verlangen nicht unbedingt schnelle, aber dafür garantierte Antwortzeiten.



**Abb. 1.3.** Bei Systemen mit einem Prozessor und mehreren Tasks muss eine Task-Umschaltung mit Sicherung des Kontextes erfolgen.

Eine erste Klassifizierung der Scheduling-Algorithmen erfolgt in nicht-verdrängendes (non-preemptive) und verdrängendes (preemptive) Scheduling. Im ersten Fall kann auch ein Prozess höherer Priorität einen laufenden Prozess nicht verdrängen; er wird erst nach Abschluss des laufenden Prozesses ausgeführt. Im zweiten Fall kann ein höherpriorer Prozess einen anderen, aktuell laufenden Prozess zum Pausieren zwingen. Der niederpriorer Prozess wird bis zum Abschluss der Bearbeitung des neuen Prozesses angehalten.

Folgende Scheduling-Verfahren sind gebräuchlich [Brosenne 08]:

<sup>9</sup> Zu Details bzgl. Tasks und Threads vgl. Kapitel 12.

*Shortest Job First*

Bei diesem einfachen Scheduling-Verfahren wird jeweils der Prozess mit der kürzesten Rechenzeit als nächstes gerechnet. Laufende Prozesse werden hierbei nicht unterbrochen (nicht-verdrängend, non-preemptive). Das Verfahren ist nicht fair, da kurze Prozesse lange Prozesse überholen können und damit bevorzugt behandelt werden. Das Verfahren ist für Echtzeitanwendungen ungeeignet.

*First Come First Served*

Wieder handelt es sich um ein relativ einfaches Scheduling-Verfahren: Die Prozesse werden gemäß ihrer Ankunftsreihenfolge dem Prozessor zugeteilt. Auch hier werden laufende Prozesse nicht unterbrochen (nicht-verdrängend, non-preemptive). Das Verfahren ist fair, da gesichert ist, dass jeder Prozess bearbeitet wird. Das Verfahren ist für Echtzeitanwendungen ungeeignet.

*Round-Robin*

Im Vergleich zu den vorangegangenen Verfahren ist das Round-Robin-Scheduling bereits vergleichsweise ausgefeilt. Bei diesem Verfahren wird die Rechenzeit in Zeitscheiben gleicher Länge aufgeteilt, die Prozesse werden in eine Warteschlange eingereiht und gem. dem Verfahren First-In-First-Out ausgewählt. Ein zu rechnender Prozess wird nach Ablauf der aktuellen Zeitscheibe unterbrochen. Er pausiert und wird erneut hinten in die Warteschlange eingefügt. Das Verfahren ist fair, die Rechenzeit wird gleichmäßig und gerecht auf die Prozesse aufgeteilt. Es ist mit Einschränkungen für Anwendungen mit weichen Echtzeitanforderungen geeignet.

*Round-Robin mit Prioritäten*

Das Round-Robin-Verfahren kann durch die Einführung von Prioritäten ergänzt werden. Hierbei werden Prozesse mit gleicher Priorität zusammengefasst in eine gemeinsame, dedizierte Warteschlange. Dann wird immer die Prozess-Warteschlange mit den höchstpriorioren Prozessen nach dem Round-Robin-Verfahren abgearbeitet. Es ist allerdings zu beachten, dass es geschehen kann, dass Prozesse nie gerechnet werden (verhungern). Eine mögliche Abhilfe ist eine Erhöhung der Priorisierung gem. der Wartezeit.

Dieses Scheduling-Verfahren, in Kombination mit Erweiterungen um das genannte Verhungern auszuschließen, stellt den Stand der Technik bei Desktop-Betriebssystemen wie Windows XP oder Linux dar.

*Statisches Echtzeit-Scheduling am Beispiel des  
Rate-Monotonic Scheduling (RMS)*

Beim statischen Scheduling wird der Schedule bereits zur Compile-Zeit für alle möglichen Tasks aufgestellt. Folgende Voraussetzungen werden nun für die weiteren Betrachtungen festgehalten:

1. Die zeitkritischen Prozesse treten periodisch auf.
2. Vor dem Bearbeiten der nächsten Periode muss die Bearbeitung der aktuellen Periode abgeschlossen sein.
3. Abhängigkeiten zwischen den Prozessen sind ausgeschlossen.
4. Die Bearbeitungszeiten sind konstant.
5. Wenn nicht-periodische Prozesse auftreten, so sind sie nicht zeitkritisch.
6. Prozesse werden verschieden priorisiert und können einander unterbrechen (es liegt ein preemptives System vor).

Hieraus lässt sich bereits eine allgemeine Bedingung formulieren. Ein Schedule existiert genau dann, wenn gilt:

$$\sum_{i=1}^n \frac{\Delta t_i}{Per_i} \leq 1 \quad (1.1)$$

Hierin ist:  $n$ : Anzahl der Prozesse,  $\Delta t_i$ : Bearbeitungszeit des  $i$ -ten Prozesses,  $Per_i$ : Periode des  $i$ -ten Prozesses.

Weiterhin stellt sich nun die Frage, wie eine sinnvolle Priorisierung vorgenommen werden kann. Das RMS-Verfahren verwendet hierzu eine Priorisierung anhand der Periode des Prozesses. Die Regel lautet: Der Prozess mit der kürzesten Periode (der höchsten Wiederholrate) erhält die höchste Priorität. Die zugrunde liegende und relativ bekannte Veröffentlichung hierzu ist [Liu 73]: „Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment.“

Die Autoren weisen nach, dass mit dieser Vorgehensweise bei einer CPU-Auslastung  $\leq 70\%$  ein Schedule garantiert werden kann. Allgemein gilt für das Existieren eines Schedules nach diesem Verfahren folgender Zusammenhang:

$$\mu = \sum_{i=1}^n \frac{\Delta t_i}{Per_i} \leq n(2^{1/n} - 1) \quad (1.2)$$

Für  $n \rightarrow \infty$  konvergiert die rechte Seite gegen 0,693. Aber auch bei höheren Auslastungen ist ein Schedule nicht unmöglich. Wenn im System z.B. alle

Periodenzeiten Vielfache der kürzesten Periode sind, so existiert auch für eine CPU-Auslastung von 100 % ein Schedule.

Der Vorteil des Verfahrens ist ein garantierter Schedule bei den genannten Bedingungen. Nachteilig ist, dass nur statisch geplant werden kann und dass bei einer Prozessorlast  $> 70\%$  eine feste Schedule-Zusage nicht mehr möglich ist. Eine typische Anwendung des Verfahrens ist die Verarbeitung vorher bekannter Audio- oder Video-Streams.

### *Dynamisches Echtzeit-Scheduling am Beispiel des Earliest Deadline First-Verfahrens (EDF)*

Beim dynamischen Scheduling wird der Schedule zur Laufzeit für die aktuell auszuführenden Tasks erstellt. Ein gängiges Verfahren hierfür ist das Earliest Deadline First-Verfahren. Hierbei wird von einem preemptiven System mit dynamischer Prioritätenverteilung ausgegangen, ansonsten sind die Voraussetzungen die gleichen wie bei RMS. Die einfache Regel lautet nun: Ausgeführt wird immer der Prozess, der aktuell die kürzeste Deadline aufweist.

Die Vorteile des Verfahrens sind, dass es einfach zu implementieren ist und den Prozessor bis zu 100 % ausnutzen kann. Ein gravierender Nachteil ist aber, dass ein korrekter Schedule nicht in allen Fällen sichergestellt werden kann.

Neben den Scheduling-Verfahren ist bei Auswahl oder Einsatz eines (Echtzeit-)Multitasking-Betriebssystems auch bei der Interprozesskommunikation besonderes Augenmerk erforderlich. Weiterhin muss auch der gemeinsame Zugriff auf Ressourcen wie angebundene Hardware oder Speicher organisiert werden. Zu weiterführenden Informationen hierzu vgl. [Marwedel 07; Brosenne 08].

### **1.4.3 Systembeispiele**

Zur Realisierung eines Echtzeit-Betriebssystems sind verschiedene Ansätze denkbar. Ein erster einfacher Ansatz ist die Modifikation des Kernels derart, dass Unterbrechungen möglich werden (preemptives Multitasking). Ergänzt mit einer Minimierung der Interrupt-Maskierungszeiten ist mit einem solchen System zumindest weiche Echtzeit gut möglich. Vertreter dieser Architektur sind: SunOS 5.x und AiX 3.0.

Eine andere Möglichkeit ist die Erweiterung eines Standard-Betriebssystems um einen preemptiven, echtzeitfähigen Mikrokern. Bei diesem aufwändigen Ansatz ist entsprechend auch eine Neu-Implementierung des Schedulers notwendig, das System kann damit aber auch für harte Echtzeitanforderungen ausgelegt werden. Vertreter: VxWorks und QNX.

Eine weitere Technik, die auch häufig angewandt wird, ist der Ersatz des Standard-Schedulers durch einen Echtzeit-Scheduler, welcher primär das Scheduling der Real-Time-Tasks organisiert und dem eigentlichen Basis-Betriebssystem nur dann Rechenzeit zugesteht, wenn die RT-Tasks abgearbeitet sind. Hier läuft das Basissystem als sog. Idle Task. Mit dieser Methode sind auch herkömmliche Systeme relativ leicht für (synchrone) Echtzeitanwendungen zu erweitern. Vertreter dieser Architektur sind beispielsweise: RTAI Linux, OSADL Realtime Linux.

Zu weiteren Details vgl. auch die weiterführende Literatur unter [Wörn 05; Marwedel 07; Yaghmour 08].

## 1.5 Software-Entwicklung

Bei der Software-Entwicklung für eingebettete Systeme sind verschiedene Vorgehensweisen möglich. Üblich, wenn auch nicht zwingend, ist die Software-Entwicklung auf einem Standard-PC mit einer leistungsfähigen IDE<sup>10</sup>. Erst nach Abschluss des Übersetzungsvorganges wird dann die Binärdatei zum Testen auf das eingebettete System, das Target, geladen. In Abbildung 1.4 ist solch ein Entwicklungssystem als Beispiel skizziert.

In diesem Beispiel wird der freie C-Compiler SDCC (Small Device C Compiler) verwendet, der auf dem GNU-Compiler basiert. Dieser sog. Cross Compiler läuft auf dem Entwicklungs-PC unter Linux oder auch Windows, erzeugt aber Maschinencode für eine zweite Plattform – im Beispiel für den 8051-Mikrocontroller.

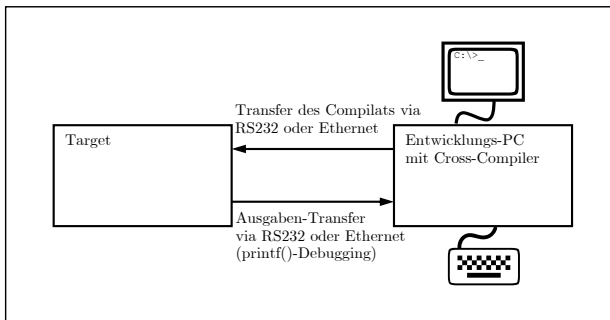
Der Aufruf erfolgt standardmäßig an der Kommandozeile, der Compiler kann aber auch in eine IDE eingebunden werden. Ein Simulator ist nicht vorhanden. Nach dem Übersetzen wird die entstandene Datei, welche den Controller-Maschinencode im standardisierten Intel-HEX-Format enthält, mit einem Tool wie beispielsweise AtmelISP seriell in den Flash-Speicher des Controllers übertragen und dort ausgeführt.

Die Vorteile eines solchen Entwicklungssystems sind die Einfachheit und die geringen Anschaffungskosten, der große Nachteil ist, dass im Falle eines Programmfehlers keinerlei Informationen zu Registerinhalten oder Speicherbelegung zugänglich sind. Die einzige Möglichkeit, an Debugging-Informationen zu gelangen, ist, in den Quelltext an sinnvollen Stellen printf()-Kommandos einzubauen.

Bei der Programmausführung überträgt der Controller dann die fraglichen Ausgaben über eine zweite serielle Schnittstelle zurück zum PC, wo sie in einem Terminal-Programm wie SerialPort oder Minicom empfangen und angezeigt

<sup>10</sup> Integrated Development Environment, integrierte Entwicklungsumgebung.

werden können. Diese asketische Art der Programmentwicklung setzt erfahrene Software-Entwickler mit großem Leidenspotenzial voraus (Abbildung 1.4).



**Abb. 1.4.** Software-Entwicklung für ein Mikrocontroller-System ohne JTAG-Debugging-Interface (sog. printf()-Debugging). Beispielsystem: Elektor 89S8252-Flashboard + SDCC-C-Compiler für die 8051-Familie.

Eine aufwändigere, aber auch ungleich informativere Art des Debuggings bieten die In-Circuit-Emulatoren (ICEs). Mit dem Begriff der In-Circuit-Emulation wurde früher – bevor die Prozessoren und DSPs spezielle Schnittstellen hierfür aufwiesen – tatsächlich der Vorgang bezeichnet, den Prozessor im Target vollständig zu ersetzen. Durch Abgriffe (sog. Bondout-Chips), die in den Entwicklungs-PC führten, konnte der PC-Prozessor den Target-Prozessor tatsächlich „emulieren“.

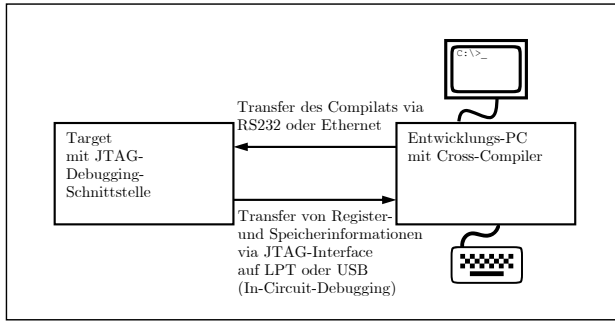
Mittlerweile tragen eingebettete CPUs oft bereits eine Debugging-Schnittstelle *on board*, das sog. JTAG-Interface.<sup>11</sup>

Über diese Schnittstelle gelangen aktuelle Informationen zur Register- und Speicherbelegung per paralleler oder USB-Übertragung zur Anzeige in den PC. Der Protokollumsetzer wird oft auch als Hardware-Emulator bezeichnet, die Bezeichnung ist aber eigentlich irreführend: hier wird das System nicht simuliert oder emuliert, sondern es wird eine Übertragung von Systeminformationen während des Betriebes ermöglicht (Abbildung 1.5).

Im Gegensatz zu den Hardware-ICEs ist hiermit keine komplette Überwachung der CPU möglich, dafür sind die erforderlichen Hardware-Komponenten aber auch wesentlich preiswerter.

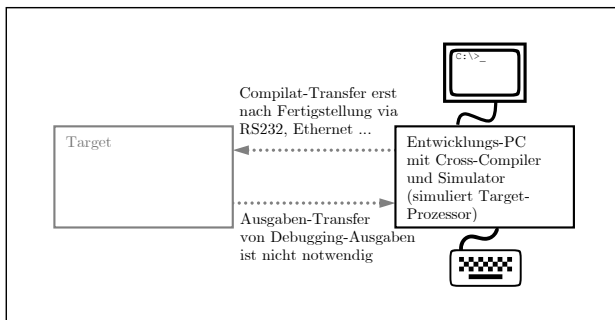
<sup>11</sup> Joint Test Action Group-Standard, ein Verfahren zum Testen und zum Debugging elektronischer Hardware direkt in der Schaltung (sog. On-Chip-Debugging, vgl. auch IEEE 1149.1).





**Abb. 1.5.** Software-Entwicklung für ein Mikrocontroller- bzw. DSP-System mit JTAG-Interface. Beispielsystem: Vision Components Smart Camera VCSBC4018 + TI C++-Compiler Code Composer + USB-JTAG-Interface.

Eine weitere Variante, Software für eingebettete Systeme zu entwickeln, ist die Verwendung eines sog. Simulators<sup>12</sup> auf dem Entwicklungs-PC. Ein solcher rein softwarebasierter Simulator bildet nicht das komplette Target-System, sondern nur dessen Ein- und Ausgaben nach. Weiterhin besitzt er die Fähigkeit, vom Cross Compiler generierten Maschinen- oder auch Zwischencode zu verarbeiten. Ein Beispiel hierzu ist das Entwicklungssystem gem. Abbildungen 1.6 und 1.7<sup>13</sup>. Dieser Simulator von der Fa. Apple ermöglicht dem Software-Entwickler ein Testen des Programms auch ohne Vorhandensein einer Target-Plattform.



**Abb. 1.6.** Software-Entwicklung für ein Mikrocontroller- bzw. DSP-System im Simulator. Beispielsystem: iPhone SDK + iPhone Simulator.

<sup>12</sup> Die Verwendung der Begriffe Simulator und Emulator ist unter Embedded-Entwicklern nicht einheitlich, hier ist Vorsicht geboten. Im vorliegenden Text werden die Begriffe entsprechend im Kontext festgelegt.

<sup>13</sup> Bildquelle: <http://flickr.com>, pena2; Lizenz: Creative Commons Attribution 2.0.

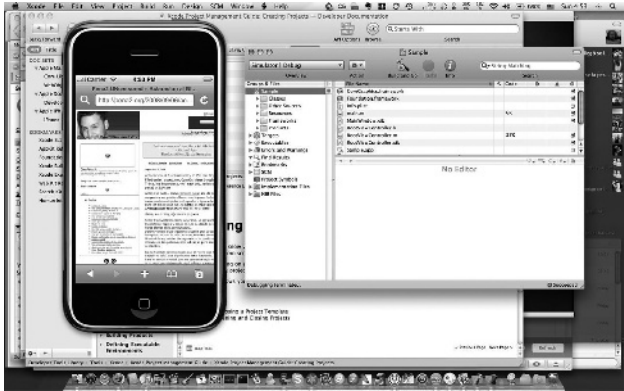


Abb. 1.7. Screenshot: SDK und Simulator zur Programmierung des iPhone.

Die Vorteile liegen auf der Hand: Der Entwicklungs-, Test- und Debugging-Zyklus wird wesentlich beschleunigt. Die Nachteile bei dieser Vorgehensweise sind, dass keine Systeminformationen zu Speicherbelegung, Registern, Programmzähler usw. verfügbar sind. Die Simulation bildet diese internen Systemdetails der Targetplattform nicht ab und wird sich daher auch nur näherungsweise wie das Target verhalten. Komplexen Problemen ist hiermit nicht beizukommen, und entsprechend ist die rein simulationsbasierte Software-Entwicklung generell fast ausgeschlossen.

Die Verwendung eines Software-*Emulators* auf dem Entwicklungs-PC vermeidet bereits einige der Nachteile des Simulators (vgl. auch Abbildungen 1.8 und 1.9). Der sog. Emulator bildet den beteiligten Prozessor vollständig mitsamt Registern und Speicher nach und ermöglicht dieserart ein sehr informatives Debugging. Ein weiterer Vorteil ist, dass trotz dieser „Hardware-Nähe“ hierfür keine zusätzliche Hardware erforderlich ist.

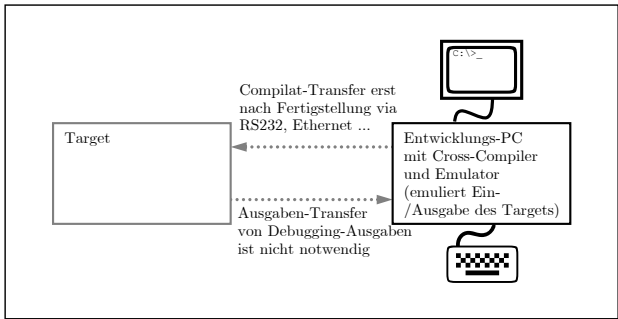


Abb. 1.8. Software-Entwicklung für ein Mikrocontroller- bzw. DSP-System im Emulator. Beispielsystem: KEIL  $\mu$ Vision2 + KEIL MCBx51 Evaluation Board.

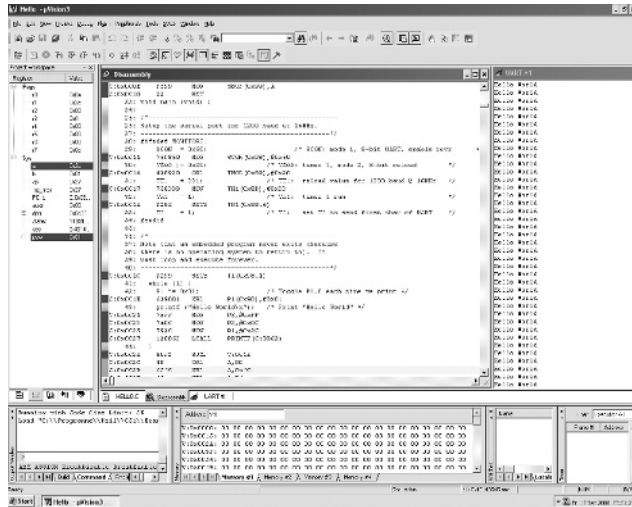


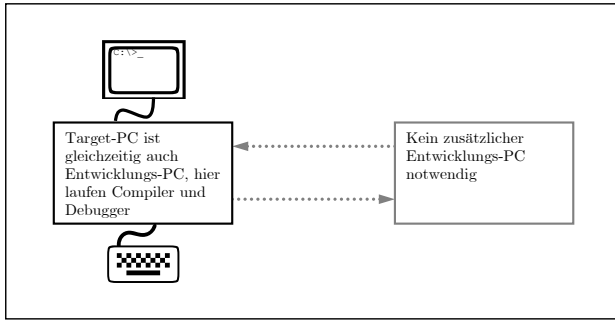
Abb. 1.9. Screenshot: SDK und Emulator KEIL µVision3.

Abschließend sei noch eine Art der Software-Entwicklung erwähnt, die erst in jüngster Zeit sinnvoll möglich geworden ist: die native Entwicklung direkt auf der Target-Plattform (Abbildung 1.10).

Mittlerweile stehen Embedded-PCs zur Verfügung, die zwar in der Rechenleistung nicht mit modernen Desktop-PCs gleichziehen können, aber dennoch eine angenehme Geschwindigkeit auch beim Betrieb einer modernen IDE bieten. Einige solche Beispiele sind NorhTec MicroClient Sr., die PC-basierte Sony-Smart-Camera-Serie und moderne Boards mit Atom- oder VIA-Nano-CPU.

Diese Plattformen bieten teilweise auch Anschlussmöglichkeiten für Netzwerk, Monitor, Tastatur und Maus und ermöglichen dieserart ein Entwickeln wie vom Desktop-PC gewohnt. Eingesetzt wird hier naturgemäß kein Cross Compiler, sondern der der Plattform zugehörige Compiler. Simulatoren, Emulatoren und JTAG-Datenübertragung entfallen, da das Debugging direkt auf der Plattform stattfindet – ein Sachverhalt, der die Fehlersuche wesentlich beschleunigt und entsprechend auch kürzere Entwicklungszeiten ermöglicht.

Die vorgestellten Möglichkeiten der Software-Entwicklung werden in der Praxis oft kombiniert angewandt. Sinnvollerweise findet ein Prototyping zuerst auf einem schnellen Entwicklungs-PC statt. Ersten Fehlern ist auch hier bereits mit den angesprochenen Simulatoren und Emulatoren beizukommen. Danach erfolgen erste Tests auf der Target-Plattform und schließlich werden hartnäckige und schwer aufzufindende Fehler per JTAG-Interface untersucht bzw. direkt mit einem Debugger auf der Target-Plattform gelöst.



**Abb. 1.10.** Software-Entwicklung für einen leistungsfähigen Embedded-PC. Beispielsysteme: NorhTec Microclient Sr., Sony Smart Camera XCI-V3.

## 1.6 Aufbau und Gebrauch des Buches

Im vorliegenden Buch wird als Betriebssystem für die Zielplattform wie auch für den Entwicklungs-PC durchgehend Linux verwendet. Hier ist es sinnvoll, sich zuerst einmal mit diesem Betriebssystem auf dem vertrauten Desktop-PC bekannt zu machen. Ein guter Einstieg gelingt mit einer Umstellung des Arbeitsplatz-PCs auf Ubuntu. Dieses Debian-basierte Betriebssystem ist besonders einfach zu installieren, zu konfigurieren und zu warten und auch mittlerweile wirklich ausgereift. DVDs mit der aktuellen Distribution finden sich fast jede Woche als Beigabe in den einschlägigen Fachzeitschriften. Das System kann für ein erstes Beschnuppern als Live-System von CD oder DVD gestartet werden. Es kann aber auch vom gleichen Datenträger eine Installation auf die Festplatte erfolgen, wobei eine evtl. bestehende Windows-Installation nicht berührt wird und weiterhin nutzbar ist. Für einen raschen und problemlosen Einstieg wurde mittlerweile zu Ubuntu auch ein Handbuch frei verfügbar gemacht, vgl. hierzu auch folgende Quellen:

<http://www.ubuntu-anwenderhandbuch.org>  
<http://www.ubuntuusers.de>

Nachdem nun bereits ein Einstieg erfolgt und auch der Umgang mit der Kommandozeile in der Konsole nicht mehr fremd ist, werden auch die Beschreibungen im vorliegenden Buch dem Leser keine Rätsel mehr aufgeben (zur Konsole vgl. auch [Linux-Kompendium 08]). Weiterhin finden sich im Anhang A auch Kurzreferenzen für die im Buch verwendeten Werkzeuge und Befehle.

Das Buch ist in drei Hauptteile gegliedert: *Im ersten Teil* werden die notwendigen Grundlagen vermittelt und die verwendeten Hardware-Plattformen vorgestellt. Weiterhin wird die Installation des Betriebssystems Linux in verschiedenen Derivaten Schritt für Schritt erläutert. *Der zweite Teil* stellt praxisnah

und ausführlich die Anpassung und Verwendung der ausgewählten Hardware-Plattformen für bestimmte Aufgaben vor: Wie wird auf die serielle Schnittstelle zugegriffen? Wie auf digitale Ein- und Ausgänge? Wie kann der Inter-IC-Bus angeschlossen und verwendet werden? Wie kann eine Kamera verwendet werden? *Im dritten Teil* respektive in den Anhängen sind Informationen zum Nachschlagen hinsichtlich des Gebrauchs der verwendeten Tools und der Installation der verwendeten Bibliotheken beigefügt. Das Buch schließt mit ausführlichen Literatur- und Sachverzeichnissen.

Parallel zum Erscheinen des Buches haben wir eine begleitende Website eingerichtet. Der Zugang kann über die Website des Springerverlages, über die Praxisbuch-Website (**praxisbuch.net**) oder auch direkt erfolgen (letzten Schrägstrich nicht vergessen):

`http://www.praxisbuch.net/embedded-linux/`

Auf dieser Website im Download-Bereich werden die Inhalte zum Buch bereitgestellt. Eine besonders aktuelle Version der Quelltext-Dateien kann alternativ auch über Subversion bezogen werden. Die URL zum SVN-Server ist auf der angegebenen Website zu finden. In Anhang F ist die Struktur der Quelltextverzeichnisse abgebildet.

## Hardware-Plattformen

### 2.1 Einführung

In diesem Kapitel werden verschiedene Hardware-Plattformen vorgestellt, die in den Folgekapiteln in Kombination mit speziellen Embedded-Linux-Varianten verwendet werden. Die Geräte stammen aus unterschiedlichen Anwendungsbereichen; so wurden die NSLU2 von Linksys sowie der WL-500gP von ASUS für Multimedia- und Netzwerk-Aufgaben entwickelt, und sind durch die weite Verbreitung sehr günstig zu beziehen. Mit dem MicroClient Jr. bzw. Sr. sind Vertreter aus dem Bereich der Micro-PCs enthalten, welche im Gegensatz zu den anderen Rechnern dieser Größe zusätzlich über eine VGA-Schnittstelle verfügen. Preislich etwas höher liegt der OpenRISC Alekto der Fa. VisionSystems, allerdings wurde dieses Gerät für den industriellen Einsatz konzipiert und erfüllt entsprechend höhere Anforderungen an Robustheit und Temperaturfestigkeit. Mit dem Mainboard D945GCLF2 der Fa. Intel wird ein Vertreter der vergleichsweise neuen Nettop-Klasse vorgestellt, die sich durch geringen Formfaktor und Stromverbrauch den Embedded-Boards nähern, zugleich aber eine wesentlich höhere Rechenleistung besitzen und vollständig x86-kompatibel sind.

Neben den Eckdaten der Geräte werden in diesem Kapitel auch Anleitungen für Erweiterungen wie serielle Schnittstellen oder I<sup>2</sup>C-Busschnittstellen vorgestellt. Eine Schaltung zur Anpassung der internen seriellen Schnittstellen an RS-232-konforme Pegel, die bei einigen Geräten benötigt wird, ist am Ende des Kapitels erklärt. Alle Modifikationen bringen einen Garantieverlust mit sich und erfolgen auf eigene Verantwortung.

## 2.2 Network-Attached-Storage NSLU2

Bei der NSLU2<sup>1</sup> von Linksys handelt es sich um ein NAS-Gerät<sup>2</sup> aus dem Consumer-Bereich. Solche Geräte dienen dazu, externe USB-Festplatten mit einer Netzwerkschnittstelle zu versehen. Aufgrund der Tatsache, dass sich auf der NSLU2 etliche verschiedene Linux-Varianten wie *Unslung*, *Debian/NSLU2* oder *SlugOS/LE* installieren lassen, ist sie in der Linux-Gemeinde sehr verbreitet. In einschlägigen Foren taucht die NSLU2 oft unter dem Namen *Slug* (deutsch: Schnecke) auf – eine Anspielung auf die im Vergleich zu PCs geringe Rechenkapazität. Die Website der *NSLU2-Linux Development Group* [NSLU2-Linux 08] bietet die wohl umfassendsten Informationen zur NSLU2, darauf aufbauenden Softwareprojekten und möglichen Hardware-Modifikationen. Die Eckdaten der NSLU2 sind in Tabelle 2.1 aufgelistet.

Architektur	ARM
Chip-Hersteller	Intel
Prozessor	XScale IXP420
CPU-Takt	266 MHz (bei älteren Versionen nur 133 MHz)
Flash	8 MB
RAM	32 MB
Ethernet	100 Mbit
USB	2× USB 2.0
Serielle Schnittstelle	1× (nicht herausgeführt)
I <sup>2</sup> C	1× (nicht herausgeführt)
Realzeituhr	X1205 (über I <sup>2</sup> C)
Preis	ab ca. 70 EUR inkl. Mwst.

**Tabelle 2.1.** Eckdaten der NSLU2.

Ältere NSLU2-Versionen wurden nur mit 133 MHz Taktfrequenz betrieben, der gleiche Prozessor läuft bei neuen Versionen nun mit 266 MHz. Zum Öffnen der NSLU2 wird das Gehäuse, wie in Abbildung 2.1 (links) angedeutet, an drei Stellen an der dunklen Gehäuseseite zusammengedrückt und dann vorsichtig auseinander gezogen, bis die Platine frei liegt. Die für eigene Erweiterungen relevanten Bauteile sind in Abbildung 2.1 (rechts) hervorgehoben.

Tabelle 2.2 zeigt die Pinbelegung des seriellen Ports auf Steckverbinder J2. Um reguläre RS-232-Pegel zu erhalten, ist eine Pegelanpassung notwendig (siehe Abschnitt 2.7). Ohne diese Erweiterungsplatine kann die serielle Schnittstelle Schaden nehmen!

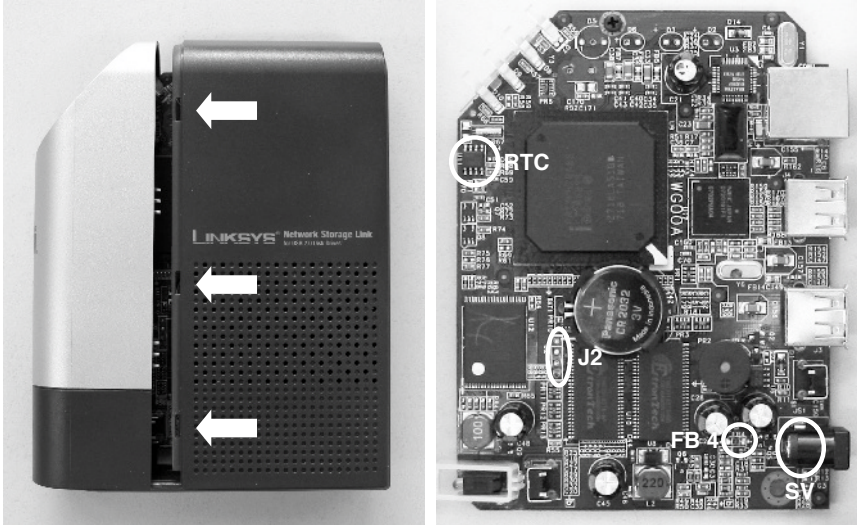
Neben der seriellen Schnittstelle bietet die IXP420-CPU 16 sog. General Purpose Input/Output-Pins (GPIO-Pins), welche zur Ansteuerung von Leuchtdi-

<sup>1</sup> Network Storage Link für USB 2.0.

<sup>2</sup> Network Attached Storage, einfach zu verwaltender Dateiserver.

Pin 1	3,3 V (weiße Markierung)
Pin 2	RX
Pin 3	TX
Pin 4	GND

**Tabelle 2.2.** Belegung des seriellen Steckverbinders J2.



**Abb. 2.1.** Druckpunkte zum Öffnen der NSLU2 (links) und freiliegende Platine (rechts) mit Echtzeituhr (RTC), serieller Schnittstelle auf Pfofenstecker (J2) und Möglichkeit zum Abgriff einer Spannungsversorgung (5V an FB4 oder SV, Masse an SV).

oden, zur Realisierung einer I<sup>2</sup>C-Schnittstelle und zur Kommunikation mit dem USB-Controller verwendet werden (vgl. Tabelle 2.3). Wird für den Anschluss eigener Komponenten ein direkter Zugang zu den GPIO-Pins gewünscht, so können einige davon zweckentfremdet werden. Alternativ lassen sich GPIO-Ports auch über einen an den I<sup>2</sup>C-Bus angeschlossenen IO-Expander nachrüsten (siehe Kapitel 9). Für alle Anwendungen, bei denen eine gewisse Latenz durch die I<sup>2</sup>C-Kommunikation akzeptabel ist, stellt dies eine einfache und sichere Lösung dar.

Zur Kommunikation mit der Realzeituhr X1205 wird über die GPIO-Pins 6 und 7 der Intel IXP420-CPU ein I<sup>2</sup>C-Master-Interface mit 100 kHz emuliert. Falls eigene I<sup>2</sup>C-Teilnehmer angeschlossen werden sollen, so bietet es sich an, die I<sup>2</sup>C-Busleitungen SDA und SCL und die 5V-Spannungsversorgung nach außen zu führen. Als Steckverbindung kann der USB-Standard verwendet werden; eine sinnvolle Belegung wird in Abschnitt 8.3.1 empfohlen.



GPIO-Pin	IXP-Anschluss	Funktion	Konfiguriert als
0	Y22	Status-LED rot (1 = An)	Ausgang
1	W21	Ready-LED grün (1 = An)	Ausgang
2	AC26	LED Disk 2 (0 = An)	Ausgang
3	AA24	LED Disk 1 (0 = An)	Ausgang
4	AB26	Summer	Ausgang
5	Y25	Power-Taste (Impuls bei Wechsel)	Eingang
6	V21	I <sup>2</sup> C SCL	Ausgang
7	AA26	I <sup>2</sup> C SDA	Tristate
8	W23	Power Off (1 = Ausschalten)	Ausgang
9	V22	PCI INTC	Eingang
10	Y26	PCI INTB	Eingang
11	W25	PCI INTA	Eingang
12	W26	Reset-Taste (0 = betätigt)	Eingang
13	V24	PCI-Reset	Ausgang
14	U22	PCI-Takt (33 MHz)	Ausgang
15	U25	Erweiterungsbustakt (33 MHz)	Ausgang

**Tabelle 2.3.** Verwendung der 16 GPIO-Pins des IXP420-Prozessors.

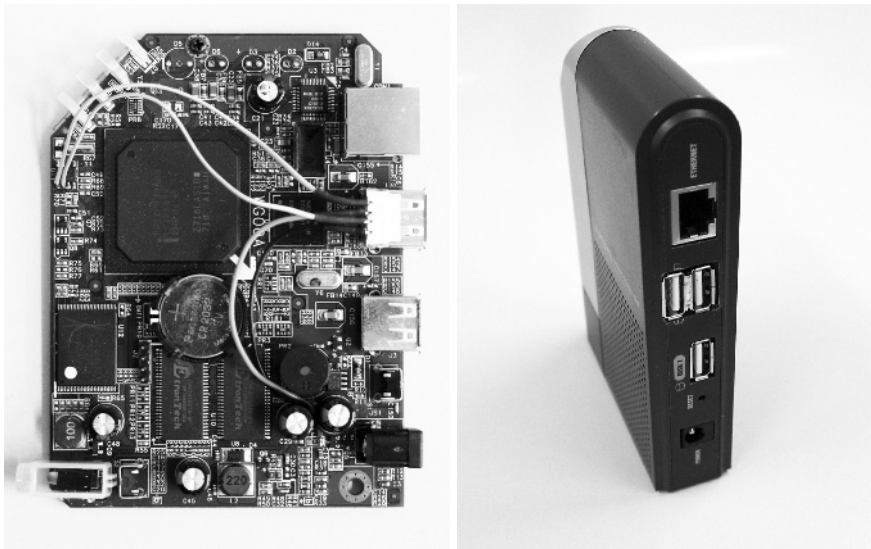
Im einfachsten Fall werden die beiden I<sup>2</sup>C-Busleitungen direkt am achtpoligen IC der Realzeituhr angeschlossen (vgl. Abbildung 2.1). Die Datenleitung SDL wird dabei mit Pin 5 verbunden (Pin unten links an der Real-time Clock (RTC)), das Taktsignal SCL mit Pin 6. Die Versorgungsmasse kann direkt am oberen Pin der Spannungsversorgungsbuchse abgegriffen werden. Die 5 V-Spannung kann ebenfalls an der Versorgungsbuchse oder an der rechten Seite des Bausteins FB4 abgenommen werden. Dieses Anschlussschema bietet den Vorteil, dass die Versorgung über den Taster der NSLU2 geschaltet wird und damit die angeschlossenen I<sup>2</sup>C-Komponenten stromlos bleiben, solange die NSLU2 abgeschaltet ist. Die NSLU2 wird in Kapitel 4 in Kombination mit einem Debian-basierten Linux verwendet.

## 2.3 WLAN-Router WL-500gP

Der WL-500g Premium<sup>3</sup> von ASUS ist zunächst ein gewöhnlicher WLAN-Router, der jedoch durch zwei USB-Schnittstellen die Möglichkeit bietet, Kartenleser, Festplatten, Webcams, Drucker oder USB-Soundkarten anzuschließen und somit auch als Embedded-Plattform interessant wird. Der Router ist im Einzelhandel bereits ab ca. 70 EUR erhältlich. Tabelle 2.4 listet die Eckdaten des Gerätes auf. Der WL-500 bietet die Hardware-Basis für die Einführung in die Linux-Distribution *OpenWrt* in Kapitel 3.

---

<sup>3</sup> Im Folgenden nur noch als WL-500 bezeichnet.



**Abb. 2.2.** Lötstellen für das Kabel für den nach außen geführten I<sup>2</sup>C-Bus mit 5V-Spannungsversorgung. Als I<sup>2</sup>C-Steckverbindung wird der USB-Standard mit einer Belegung gemäß Kapitel 8.3.1 verwendet. Der entsprechende Steckverbinder kann an den USB-Anschluss von Disk 2 gelötet werden.

Architektur	MIPS
Chip-Hersteller	Broadcom
Prozessor	Broadcom BCM94704
CPU-Takt	266 MHz
Flash	8 MB
RAM	32 MB (teilweise nur 16 MB freigeschalten)
Wireless	MiniPCI Broadcom 802.11b/g BCM4318
Ethernet	Robo switch BCM5325
USB	2× USB 2.0
Serielle Schnittstelle	2× (nicht herausgeführt)
Preis	ab ca. 70 EUR inkl. Mwst.

**Tabelle 2.4.** Eckdaten des WL-500.

Zur Nutzung der seriellen Schnittstellen oder zum Austausch der WLAN-Karte muss das Gehäuse geöffnet werden. Dazu werden die vier schwarzen Kappen an der Unterseite abgenommen und die darunter liegenden Schrauben herausgedreht. Die beiden Gehäuseschalen lassen sich dann leicht auseinander ziehen.

Der WL-500 besitzt zwei serielle Schnittstellen, die sich auf den acht Lötunkten am linken Rand der Platine befinden (J10 in Abbildung 2.3). Die Schnittstelle hat die in Tabelle 2.5 gezeigte Belegung. Eine JTAG-Schnittstelle<sup>4</sup>

<sup>4</sup> Vgl. Abschnitt 1.5.

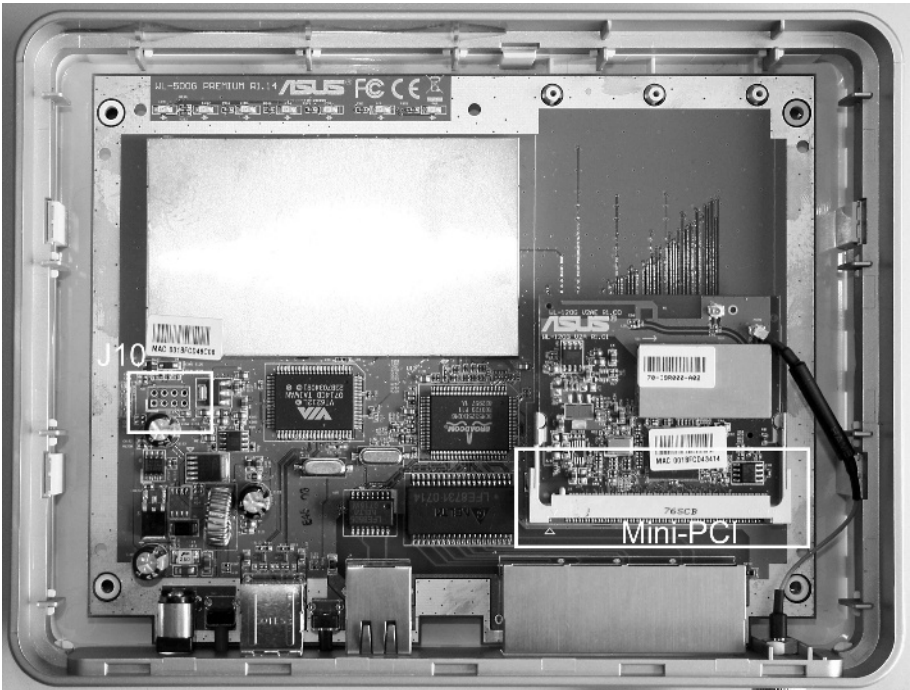


Abb. 2.3. Hauptplatine des ASUS-Routers WL-500.

ist übrigens nicht vorhanden, auch deswegen sollte beim Aufspielen neuer Firmware sorgfältig vorgegangen werden.

RX.1	TX.1	GND	RESET
RX.0	TX.0	3,3 V	—

**Tabelle 2.5.** Belegung des seriellen Steckverbinders (J10 in Abbildung 2.3), der eckige Lötpoint ist Pin 1 und gehört zu RX.0.

An den seriellen Schnittstellen liegen Logikpegel mit nur 3,3 V Maximalspannung. Entsprechend muss ein Schnittstellentreiber verwendet werden, um eine serielle Schnittstelle gemäß dem RS-232-Standard zu erhalten. Eine Schaltung dafür wird in Abschnitt 2.7 vorgestellt. Die Wireless-LAN-Schnittstelle ist über den Mini-PCI-Steckplatz realisiert. Hier wird werksseitig eine Platine mit einem Controller vom Typ BCM4318 verbaut, für den aber kein OpenWrt-Treiber für Linux-Kernel-Version 2.6 existiert. Durch Austausch mit einer Atheros-MiniPCI-Karte, für die ein Open-Source-Treiber existiert, kann Wireless LAN mit Kernel 2.6 betrieben werden. Ein I<sup>2</sup>C-Interface oder freie GPIO-Pins sind nicht auf der Platine vorhanden, können aber leicht über einen IOWarrior-Baustein nachgerüstet werden. Vgl. hierzu auch Abschnitt 3.6.

## 2.4 MicroClient Jr. und Sr.

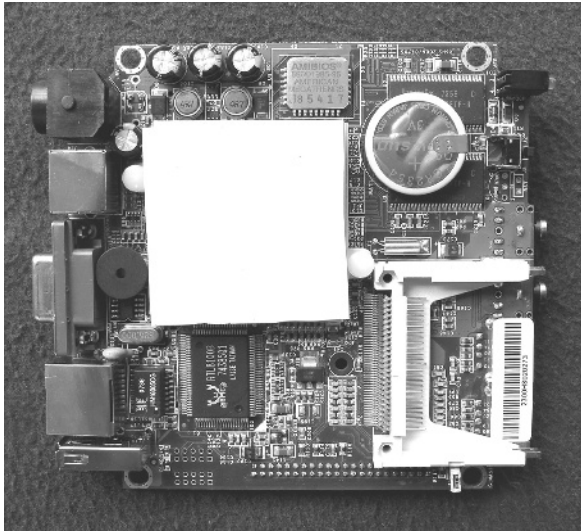
Bei den MicroClient-Jr.- und Sr.-PCs des thailändischen Distributors NorhTec handelt es sich um kleine, preiswerte und lüfterlose Thin Clients, die auch als Embedded PCs eine gute Figur machen (Abbildung 2.4). Im nachfolgenden Text wird näher erläutert, wie die PCs sinnvoll eingesetzt werden, die technischen Daten werden vorgestellt und es wird untersucht, ob der annoncierte Preis des Distributors bei einer Bestellung im Direktvertrieb wirklich realistisch ist.

Zuallererst ist interessant, dass der kleine PC baugleich unter mehreren Bezeichnungen im Handel ist: MicroClient Jr., eBox-2300, Aleutia E2, wobei die meisten Informationen über eine Produktsuche nach „eBox“ zu finden sind. Hier wird man dann auch hinsichtlich deutscher Lieferanten fündig (vgl. Anhang E). Wir haben allerdings zwei PCs direkt bei NorhTec bestellt, da das Produkt dort trotz Porto- und Zollkosten noch etwas preisgünstiger ist. Der PC ist mit oder ohne vorinstallierter CF-Card erhältlich. Wer hier noch ein wenig Geld sparen möchte, der kann auch eine günstige CF-Card besorgen und selbst das Betriebssystem aufspielen. Die genaue Vorgehensweise wird in Kapitel 6 beschrieben.

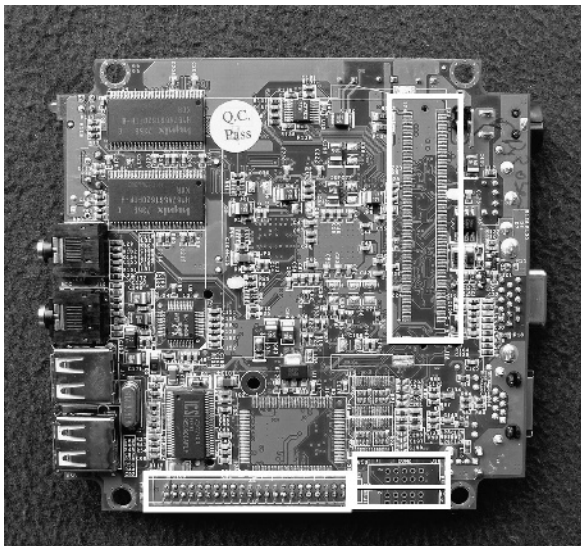


**Abb. 2.4.** Eine Handvoll Computer – der Norhtec MicroClient Jr. bzw. Sr. (die Rechner sind mechanisch identisch).

Bevor wir den PC in Betrieb genommen haben, haben wir ihn zuerst einmal für eine intensivere Untersuchung geöffnet (Abbildungen 2.5, 2.6). Das ist nicht ganz einfach, und die Gefahr des Garantieverlustes bzw. der Zerstörung ist



**Abb. 2.5.** Basisplatine des MicroClient Jr. – Oberseite. Gut zu erkennen ist der CF-Card-Adapter und das Silikonkissen auf dem Prozessor zur thermischen Ankopplung an das Gehäuse.



**Abb. 2.6.** Basisplatine des MicroClient Jr. – Unterseite. Markiert sind die zwei seriellen Schnittstellen, die IDE- und die Mini-PCI-Schnittstelle.

nicht von der Hand zu weisen. Wer es dennoch wagen will: Zuerst sind alle zugänglichen Schrauben zu lösen (vier im Deckel, zwei an der VGA-Buchse), dann ist die Blende um die Netzwerk- und USB-Buchsen vorsichtig ein wenig hochzuhebeln. Nach und nach kann dann mit viel Gefühl die u-förmige seitliche Gehäuseschale ausgeklinkt und der PC zerlegt werden (vgl. Abbildungen). Zu beachten ist, dass der Prozessorkühlkörper über ein Silikonkissen am Deckel klebt. Auch diese Verbindung kann vorsichtig von der Seite mit einem Schraubenzieher auseinandergehebelt werden. In der Vergleichstabelle 2.6 sind die wichtigsten technischen Daten der drei MicroClient-Typen zusammengestellt.

Auch die Daten der mittlerweile erhältlichen Derivate MicroClient Jr. SX und MicroClient Sr. sind dort aufgeführt. Diese tauchen oft ebenso wie der Jr. baugleich unter anderen Namen auf, und so listet sie der Distributor Epatec unter den Bezeichnungen eTC-2300, eTC-2500 usw. (vgl. Anhang E).

Über eBay wird der PC, u. U. unter der Bezeichnung „eBox“, häufig als Neuware angeboten; die Preise bewegen sich dort um die 170 EUR (inkl. Mwst.) für den Jr. Als würdigen Nachfolger für den mittlerweile abgekündigten Jr. sollte auf jeden Fall nur der Sr. in Betracht kommen, da der MicroClient Jr. SX zwar nochmals ein wenig günstiger ist, aber auch einen gravierenden Nachteil aufweist: Er besitzt keine FPU und ist entsprechend ein gutes Stück langsamer. Die Sr.-Version mit 1 GByte RAM liegt preislich bei Direktbezug von NorhTec um die 250 EUR inkl. Mwst., Versand und Zoll.

Der kleine PC ist optimal für einen Betrieb als sog. Thin Client geeignet. Das eigentliche Rechnen geschieht dann auf einem leistungsfähigen Server bzw. im Netz, und der Client ist nur noch für Ein- und Ausgabe zuständig (früher nannte man dies Terminal). Eine weitere typische Anwendung des kleinen Rechners ist – angebracht auf der Rückseite eines TFT-Monitors – die Verwendung als sog. Kiosk-PC. Typischerweise läuft auf solch einem Kiosk- oder Info-Terminal ein schlanker Browser, der Informationen zur Firma, zu Fahrplänen oder Sonstiges zeigt und verriegelt ist hinsichtlich weitergehender Benutzerzugriffe. Das funktioniert grundsätzlich, es sind aber bei Einsatz des MicroClient Jr. keine Geschwindigkeitsrekorde zu erwarten. Mit einem leistungsfähigen Webbrowser wie z. B. Firefox geht das System schon etwas in die Knie. Hier können sich Experimente mit schlankeren Browsern wie Dillo, Netsurf, Links2, Opera, Arora, Modora oder Conkeror als lohnenswert erweisen.

Eine andere naheliegende und für uns auch interessantere Anwendung ist die Verwendung als robuster Embedded-PC. Die Anbindung der Aktoren oder Sensoren kann hierbei über USB, über eine USB-I<sup>2</sup>C-Kopplung oder auch über die serielle Schnittstelle erfolgen. Weiterhin sind auch spezielle MicroClient-Typen mit digitalen Ein- und Ausgängen erhältlich.

Grundsätzlich eignen sich die kleinen Rechner gut für Embedded-Anwendungen. Für professionelle Anwendungen in der Industrie fehlen allerdings Details wie erweiterter Temperaturbereich, Hutschienen-Aufnahme,

Produktbezeichnung	MicroClient Jr. (eBox-2300)	MicroClient Jr. SX (eBox-2300SX)	MicroClient Sr. (eBox-4300)
CPU Chipsatz	SiS 550 Vortex86	MSTi PSX-300 Vortex86SX	VIA Eden VIA CX700M Ultra Low Voltage
MHz	200 MHz	300 MHz	500 MHz
Floating Point Unit	Ja	Nein	Ja
Kompatibel zu	Pentium MMX	486SX	VIA C7
BogoMIPS	400	122	ca. 1000
RAM, eingelötet	128 MB SD-RAM	128 MB DDR2-RAM	512 MB DDR2, max.1 GB
Grafik	SiS 550 PCI/AGP	XGI Volari Z7 / Z9S	VIA UniChrome 2D/3D Grafik mit MPEG4/WMV9 Decoding Accelerator
Grafik: Shared Memory	8 MB	32 MB	128 MB
Audio	SiS 7019; AC97 CODEC, compliant with AC97 V2.1, Buchsen für MIC-in und Line-out	—	AC97 (VIA Vinyl VT1708)
USB	3x USB 1.1	3x USB 2.0	3x USB 2.0
Ethernet	Realtek RTL8100 10/100 Base-T; RJ45	RDC R6040 10/100 Base-T; RJ45	Realtek RTL8100B 10/100 Base-T; RJ45
Festspeichermedium	CF-Card (nicht im Preis inbegriffen)	CF-Card (nicht im Preis inbegriffen)	CF-Card (nicht im Preis inbegriffen)
Schnittstellen	VGA, 3x USB, PS/2, IDE, RJ45, Audio: Mic-in + Line-out, Mini-PCI-Pads auf Platine, aber kein Steckverb., dito: 2x RS232	VGA, 3x USB, PS/2, 44pin IDE, RJ45	VGA, 3x USB, PS/2, E-IDE, RJ45, Audio: Mic-in + Line-out; 2x RS232
Mögliche Erweiterungen	24-Bit-GPIO, 2x RS232, WLAN (für Mini-PCI)	2x RS232, Mini-PCI, WLAN (für Mini-PCI), 24-Bit-GPIO, HDD-Support	WLAN (für Mini-PCI), HDD-Support
Leistung des externen Netzteils; tatsächliche Leistungsaufnahme	15 Watt 3 A @ 5 VDC	15 Watt 1080 mA @ 5 VDC	15 Watt 1,8 A @ 5 VDC (mit HDD und CD: 5 A @ 5VDC)
Abmessungen und Gehäuse-Formfaktor	115 mm x 35 mm x 115 mm (VESA-100)	115 mm x 35 mm x 115 mm (VESA-100)	115 mm x 35 mm x 115 mm (VESA-100)
Gewicht	500 g	500 g	500 g
Besonderheiten	Lüfterlos, <i>Entweder</i> Mini-PCI <i>oder</i> HDD möglich; Vorsicht: Gerät ist abgekündigt (bei NorhTec noch bis zum Jahreswechsel erhältlich)	Lüfterlos, <i>keine</i> FPU, <i>Entweder</i> Mini-PCI <i>oder</i> HDD möglich;	Lüfterlos, <i>Entweder</i> Mini-PCI <i>oder</i> HDD möglich
OS-kompatibel	Linux, Windows CE	Spezielle Linux-Distributionen mit FPU-Emulation (angepasstes Puppy Linux erhältlich); Windows CE 5.0	Linux, Windows CE, Windows XP, Windows XP Embedded
Preis	ab ca. 190 EUR inkl. Mwst.	ab ca. 145 EUR inkl. Mwst.	ab ca. 250 EUR inkl. Mwst.

Tabelle 2.6. Vergleichstabelle zu den drei MicroClient-Derivaten.

Spezifikation der Schutzart (IPxx) und Feldbusschnittstellen. Weiterhin soll auch nicht verschwiegen werden, dass die Hardware-Anbindung nicht immer ganz unproblematisch ist. Als Beispiel sei die Inbetriebnahme der WLAN-Schnittstelle des entsprechenden 2300er-Derivates unter Linux genannt. Dies scheint im Moment nur relativ umständlich und wenig elegant per NDIS-Wrapper möglich (vgl. auch Abschnitt 10.3). Im Gegenzug sind die kleinen Rechner aber auch um einiges preisgünstiger als professionelle Industrie-PCs.

Fazit: Die kleinen NorhTec-Embedded-PCs haben nicht umsonst mittlerweile eine erstaunliche Verbreitung gefunden. Sie sind hinsichtlich Baugröße und Schnittstellenumfang im Verhältnis zum Kostenaufwand kaum zu schlagen, in unterschiedlichen Derivaten erhältlich (Anhang E: Bezugsquellen) und haben einen gewissen Sympathiebonus, weil sie einfach hübsch anzuschauen sind.

Auf der ersten Generation läuft zwar schon ein komplettes, schlankes Linux, aber für Standard-Anwendungen wie Open Office sind die MicroClient-Jr.-PCs zu langsam. Anders sieht die Sache bei der neuen MicroClient-Sr.-Generation aus: Mit dem vierfachen Speicher und der vierfachen Taktfrequenz erschließen sich ganz neue Anwendungen. Interessant ist auch, dass der MicroClient Sr. bzw. der eBox-4300 mittlerweile als Standardplattform für den Imagine-Cup-Wettbewerb der Firma Microsoft ausgewählt wurde. Auch hierüber wird die Verbreitung sicherlich weiter zunehmen.

Abschließend sei noch angemerkt, dass im Netz Gerüchte kursieren, wonach die kleinen PCs mit bestimmten BIOS-Einstellungen getuned werden können. Wir haben alle verfügbaren, vermeintlich geschwindigkeitsrelevanten Modifikationen getestet (Power Management enabled/disabled, CPU Ratio, NorthBridge Top Performance, ...), konnten aber keinen nennenswerten Unterschied feststellen. Für die Messung kam hierbei das (nachinstallierte) Tool HardInfo zum Einsatz.

## 2.5 OpenRISC Alekto

Der OpenRISC Alekto der Firma VisionSystems ist ein Embedded-Rechner auf Basis eines ARM9-Prozessors [Vision Systems 08]. Eine große Zahl von Schnittstellen wie LAN, USB, Seriell (RS-232, RS-422 und RS-485), I<sup>2</sup>C und GPIOs, ein möglicher Einsatz im Temperaturbereich von  $-10^{\circ}\text{C}$  bis  $+65^{\circ}\text{C}$  sowie eine robuste Ausführung machen den Alekto für den industriellen Einsatz interessant. Tabelle 2.7 zeigt die wichtigsten Eckdaten. Der Alekto lässt sich auf einer DIN-Schiene montieren und so einfach bspw. in Schaltschränken einbauen. Die eingesteckte CF-Karte wird über einen IDE-Adapter als Festplatte angesprochen und kann mit vorinstalliertem Linux betriebsbereit erworben werden.



Architektur	ARM
Prozessor	ARM9 32-bit RISC CPU
CPU-Takt	166 MHz
Flash	4 MB
RAM	64 MB SDRAM
Wireless	MiniPCI (optional)
Ethernet	2×
USB	2× USB 2.0
Serielle Schnittstelle	2× RS-232/422/485
I <sup>2</sup> C	1×
CF-Slot	1× (True IDE)
Digital I/O	8×
Preis	ca. 240 EUR inkl. Mwst.

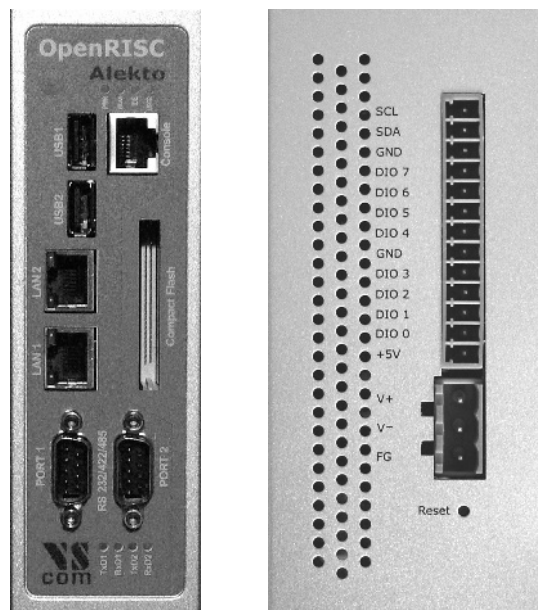
**Tabelle 2.7.** Eckdaten des OpenRISC Alekto.

Als Betriebssystem wird ein vollständiges Debian GNU/Linux für ARM mitgeliefert, sodass z. B. eine Nutzung als Web-, E-Mail-, oder Drucker-Server möglich ist. Genauso lassen sich aber auch eigene Anwendungen bzw. Steuerungsaufgaben in der Automatisierungstechnik umsetzen. Kapitel 5 gibt eine Einführung in das Betriebssystem und die Software-Entwicklung für Alekto. Abbildung 2.7 zeigt den Rechner in der Frontansicht und die Belegung des Steckverbinders an der Oberseite. Da alle verfügbaren Schnittstellen inkl. GPIOs und RS-232 bereits nach außen geführt wurden, ist ein Öffnen des Gehäuses bei diesem PC nicht notwendig.

## 2.6 Mini-ITX-Mainboard D945GCLF2 mit Dual-Core Atom CPU

Mit dem D945GCLF2 bringt Intel ein preisgünstiges Mainboard auf Basis eines Dual-Core Atom-Chips auf den Markt (vgl. Abbildung 2.8). Auch wenn der PC-Markt schnelllebig ist und schon bald mit Nachfolgern dieses Produktes gerechnet werden muss, so soll dieses Board als Vertreter der aufkommenden sog. Nettop-Klasse vorgestellt werden. Das Mainboard befindet sich in Hinblick auf die umfangreiche Ausstattung und die vergleichsweise hohe Rechenleistung an der Schwelle zu herkömmlichen PC-Mainboards – aufgrund des kleinen Formfaktors von nur  $17 \times 17 \text{ cm}^2$  lässt es sich aber auch den Embedded-Systemen zuordnen.

Auf dem Mainboard wird nicht etwa die CPU mit Kühlkörper und Lüfter gekühlt, sondern der Northbridge-Chipsatz. Die CPU befindet sich unter dem passiven Kühlkörper auf der linken Seite und nimmt laut Hersteller nur eine Leistung von ca. 8 Watt auf. Der Umstand, dass der Chipsatz anscheinend



**Abb. 2.7.** OpenRISC-Alekto in der Frontansicht mit Schnittstellen für LAN, USB, RS-232 und Konsole (links), und die Belegung des Steckverbinders an der Oberseite mit Spannungsversorgung, I<sup>2</sup>C-Bus und GPIOs (rechts).



**Abb. 2.8.** Rechner auf Basis eines Mini-ITX-Mainboard D945GCLF2 von Intel mit Dual-Core Intel Atom 330 Prozessor.

mehr Strom verbraucht als die CPU, ist für zukünftige Boards noch verbesserungswürdig. Dennoch verbraucht ein Komplettsystem mit 2,5“-Festplatte nicht mehr als 30 Watt und lässt sich durch Austausch des Northbridge-Kühlers auch komplett passiv kühlen.

Für den Einsatz des Boards bieten sich aufgrund der umfangreichen Ausstattung vielfältige Möglichkeiten (vgl. auch die Eckdaten in Tabelle 2.8): Als leiser Media-PC im Wohnzimmer, stromsparender Homeserver oder leistungsfähiges Embedded-System für dedizierte Bildverarbeitungs- oder Steuerungsaufgaben. Auch wenn man keine Rekorde erwarten darf, so lief Ubuntu im Test sehr akzeptabel, allein die Speicherausstattung von 1 GB scheint etwas knapp bemessen.

Modell	Intel D945GCLF2
Bauart	Mini-ITX
Prozessor	Intel Atom 330, 2× 1.60 GHz
Grafik	Intel GMA 950
RAM	1× DDR2 533/667 MHz
Video	VGA, S-Video
Audio	Line-in, Line-out, Mic
PCI	1×
Ethernet	1× 1000 Mbit
USB	6× USB 2.0
Serielle Schnittstelle	1× RS-232
Parallele Schnittstelle	1×
Sonstige Anschlüsse	2× SATA, 1× IDE, 2× PS/2
Spannungsversorgung	24-Pin-ATX, P4-Stecker
Abmessungen	17×17 cm <sup>2</sup>
Preis	ca. 80 EUR inkl. Mwst.

**Tabelle 2.8.** Eckdaten des D945GCLF2-Mainboards.

Aufgrund der altgedienten Schnittstellen wie RS-232 oder Parallelport eignet sich das Board außerdem, um eigene Hardware-Entwicklungen anzuschließen. Auch wenn diese über USB-Adapter oder IOWarrior-Platinen problemlos nachzurüsten sind, so ist der unmittelbare Zugriff auf die Peripherie über Speicheradressen eine Voraussetzung für viele echtzeitkritische Anwendungen.

In Kapitel 12 wird dieses Board mit einer Echtzeiterweiterung für die hochgenaue Takterzeugung über die parallele Schnittstelle genutzt. Dabei bedeutet die Verwendung der x86-Architektur einen großen Vorteil hinsichtlich der notwendigen Kernel-Patches und der Verfügbarkeit von hochauflösenden Zeitgebern (High Resolution Timers).

Bei dem Board handelt es sich aus Software-Sicht um ein herkömmliches PC-Mainboard, auf welchem ein Standard-Ubuntu-Linux aufgesetzt werden kann.

Entsprechend ist dem D945GCLF2 kein separates Kapitel zur Inbetriebnahme gewidmet.

## 2.7 Pegelanpassung für die RS-232-Schnittstelle

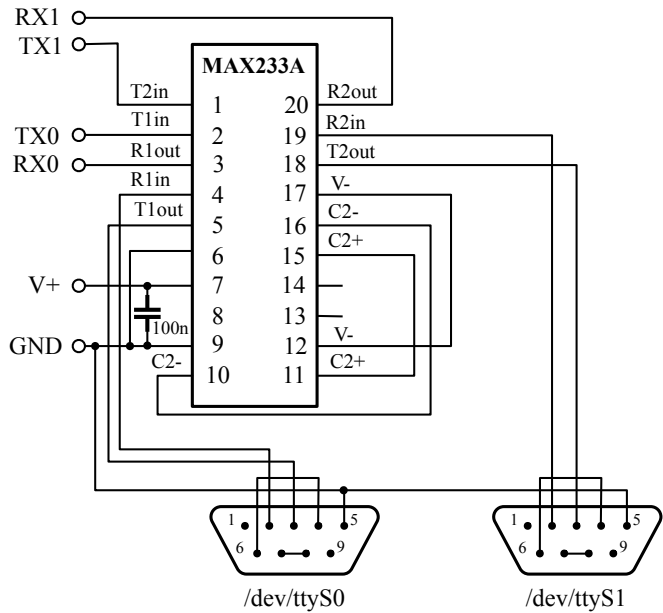
In einigen der vorgestellten Geräte sind serielle Anschlüsse zwar intern verfügbar, jedoch nicht nach außen geführt. Diese Schnittstellen werden vom Kernel erkannt und sind üblicherweise als `/dev/tts/0` und `/dev/tts/1`, bzw. `/dev/ttyS0` und `/dev/ttyS1` in das System eingebunden.

Die seriellen Schnittstellen sind für den normalen Anwender eher unwichtig, für den Embedded-Entwickler aber umso interessanter, da sie oft zu Debugging-Zwecken genutzt werden. Die Schnittstellen arbeiten mit Logikpegeln von nur  $\pm 3,3$  bzw.  $\pm 5$  V – zu wenig für eine Schnittstelle nach RS-232-Standard mit  $\pm 12$  V. Um eine vollwertige Schnittstelle zu erhalten, ist eine Pegelanpassung notwendig, welche leicht über einen MAX233-Wandlerbaustein realisiert werden kann (vgl. Tabelle E.1).

Hier stellt sich die Frage, warum nicht einfach ein USB-RS-232-Adapter verwendet werden kann; der in vielen Konvertern (vgl. Tabelle E.1) eingesetzte Chip PL2303 wird unter Debian und OpenWrt problemlos unterstützt. Gegenüber der USB-Variante bietet die Verwendung einer echten seriellen Schnittstelle jedoch den Vorteil, dass sie als Terminal dienen kann um bspw. die Meldungen während des Bootvorganges auszugeben oder auch um eine falsch konfigurierte Netzwerkverbindung über einen Telnet-Login zu reparieren. Darüber hinaus ist die Latenz geringer als beim Umweg über die USB-Schnittstelle – gerade für Echtzeitanwendungen kann dies ein wichtiges Argument sein.

Abbildung 2.9 zeigt eine mögliche Adapterschaltung mit einem MAX233-Wandlerbaustein. Bei einer Versorgungsspannung von 3,3 V werden die RS-232-Pegel auf ca. 6 V angehoben, bei einer Versorgung von 5 V auf 10 V.

An dieser Stelle sei noch angemerkt, dass die in der Linksys NSLU2, im ASUS WL-500 oder in den Norhtec MicroClients integrierten Schnittstellen keine Signale für Hardware-Handshaking zur Verfügung stellen. Die Leitungen DTR und DSR (Pins 4 und 6), sowie RTS und CTS (Pins 7 und 8) werden deshalb im Stecker verbunden.



**Abb. 2.9.** Adapterschaltung zur Realisierung zweier vollwertiger RS-232-Schnittstellen (9-pol. Sub-D Stecker) auf Basis eines MAX233-Wandlerbausteins. Die Versorgungsspannung kann 3,3 V oder 5 V betragen.

## OpenWrt auf dem WLAN-Router WL-500g Premium

### 3.1 Einführung

OpenWrt wurde ursprünglich als Linux-Distribution für WLAN-Router entwickelt, wird mittlerweile aber auch auf anderen Embedded-Geräten wie z. B. NAS<sup>1</sup> verwendet [OpenWrt 08]. Im Gegensatz zur klassischen Firmware, wie sie auf den meisten dieser Geräte verwendet wird, war das Entwicklungsziel der OpenWrt-Gemeinde ein flexibles Betriebssystem mit Dateisystem und Paket-Management, um das Gerät entsprechend der eigenen Verwendung konfigurieren zu können. Anders als bei umfangreichen Distributionen wie Debian wurde ein minimalistischer Ansatz verfolgt. Dies geschah mit dem Ziel, das komplette System im On-Board-Flash-Speicher des Gerätes unterbringen zu können (typisch: vier bis acht Megabyte). Durch die mittlerweile verfügbare große Anzahl an Paketen wird OpenWrt nicht mehr nur für die ursprüngliche Aufgabe als Router-Software verwendet, sondern vermehrt auch für die Bereitstellung von Diensten wie Web- und Drucker-Server, Dateifreigaben und Steuerungsaufgaben eingesetzt.

Ein Nachteil des minimalistischen Konzeptes ist die Tatsache, dass die Software-Entwicklung nicht mehr nativ auf dem Gerät erfolgen kann, sondern dass ein zusätzlicher Entwicklungs- bzw. Host-Rechner verwendet werden muss.

Die Entwicklung erfolgt entsprechend mit einem PC-basierten x86-System für ein anderes Zielsystem. Im Falle des WL-500 ist das die MIPSel-Architektur.<sup>2</sup> Die eigene Erstellung der für einen solchen Prozess notwendigen Toolchain ist eine komplexe Aufgabe und nur erfahrenen Nutzern zu empfehlen. Hier

---

<sup>1</sup> *Network Attached Storage*, Festplattenspeicher mit Netzwerk-Interface.

<sup>2</sup> *Microprocessor without Interlocked Pipeline Stages*, Mikroprozessor ohne Pipeline-Sperren. Der Zusatz *el* zeigt an, dass als Byte-Reihenfolge *Little Endian* verwendet wird.

kommt das OpenWrt-Build-System ins Spiel (im Folgenden als OpenWrt BS bezeichnet), welches die komplette Toolchain erstellt und alle dafür notwendigen Schritte wie Download, Patchen und Übersetzen durchführt.

Das OpenWrt BS enthält zudem eine Liste frei verfügbarer Software, die heruntergeladen, als OpenWrt-Paket gebaut und optional in den Kernel integriert werden kann. Um Software wie *dnsmasq* oder *wireless-tools* übersetzen zu können, müssen in der Regel sog. *Patches* eingespielt werden. Hiermit werden geringe Änderungen im Quelltext vorgenommen, um die Besonderheiten der Zielarchitektur zu berücksichtigen. Auch diese Aufgaben werden vom OpenWrt BS übernommen. Ebenfalls enthalten ist ein Werkzeug, um das Firmware-Image zu erstellen. Hierbei wird der fertiggestellte Kernel mit den notwendigen Modulen in eine Binärdatei verpackt und in den Flash-Speicher des Gerätes übertragen.

Das OpenWrt BS ist für die spätere Software-Entwicklung unabdingbar, und entsprechend sollte bereits die Firmware unbedingt mit dem OpenWrt BS selbst erstellt werden. Ein vorgefertigtes WL-500-OpenWrt-Image aus dem Internet enthält zudem nicht alle der möglicherweise gewünschten Kernel-Pakete.

In Abschnitt 2.3 wurde bereits die Hardware des ASUS WL-500 besprochen. Im vorliegenden Kapitel wird nun die OpenWrt-Toolchain eingerichtet, die dann unter Ubuntu bei der Programmierung eigener Applikationen Anwendung findet. Im Anschluss wird die Erstellung eigener OpenWrt-Pakete erklärt. Im letzten Abschnitt 3.6 wird auf die Anbindung eines IO-Warrior-Bausteines [Codemercs 08] eingegangen, um den WL-500 über USB um zusätzliche IOs, Schnittstellen für LC-Displays und einen I<sup>2</sup>C-Bus zu erweitern.

## 3.2 Einrichtung des OpenWrt-Build-Systems

In diesem Abschnitt wird die Einrichtung des OpenWrt BS anhand des ASUS WL-500 vorgestellt. Für die Installation auf anderen Plattformen wie Linksys WRT54G oder Linksys NSLU2 werden bei abweichendem Vorgehen an den entsprechenden Stellen Hinweise gegeben.

Im weiteren Ablauf ist die Versionsverwaltung Subversion (SVN) erforderlich. Subversion ist ein freies Tool zur Verwaltung und Versionierung von Quelltexten. Es kann, falls auf dem Debian-Host-System nicht vorhanden, mit folgendem Befehl nachinstalliert werden:

```
$ sudo apt-get install subversion
```

Liefert ein Aufruf von `svn` eine Rückmeldung, so ist Subversion bereits installiert. Die Quelltexte der aktuellen OpenWrt-Version 7.09 *Kamikaze* können

nun ausgecheckt und in unser Standardverzeichnis `<openwrt-dir>` heruntergeladen werden:

```
$ svn co https://svn.openwrt.org/openwrt/tags/kamikaze_7.09 <openwrt-dir>
```

Für Anwender, die bisher noch keine Erfahrungen mit Subversion gesammelt haben, findet sich unter [Subversion 08] eine freie Dokumentation. Im Vergleich zur Vorgängerversion *7.07 Whiterussian* wurde das Build-System komplett überarbeitet, sodass die Anleitungen in diesem Kapitel dafür nicht mehr 1:1 anwendbar sind. Für zukünftige Versionen kann sich der Pfad ändern, dies ist ggf. auf der OpenWrt-Website [OpenWrt 08] nachzulesen. Nach Ausführung der oben genannten Befehle ist nun das komplette OpenWrt BS auf den Rechner geladen. Während des nun folgenden Prozesses wird die Toolchain erstellt, um unter einer x86-Architektur Binärdateien für Broadcom-Prozessoren der Reihen BCM947xx/953xx erzeugen zu können. Neben etlichen weiteren Architekturen werden auch die Intel-XScale IXP4xx-Bausteine<sup>3</sup> und die ebenfalls verbreiteten AVR32-Prozessoren von Atmel unterstützt. Mithilfe der erstellten Toolchain wird nun der Linux-Kernel für das Zielsystem erzeugt, es werden die notwendigen Pakete gebaut und das Firmware-Image wird erstellt. Zunächst muss im neu erstellten Verzeichnis `<openwrt-dir>` folgender Befehl eingegeben werden:

```
$ make menuconfig
```

Hiermit wird die Zielarchitektur festgelegt und es wird vorgegeben, welche Pakete gebaut werden sollen (in den Kernel integriert oder als Modul zum Nachladen). U. U. müssen an dieser Stelle noch die in der Datei `<openwrt-dir>/README` gelisteten Pakete nachinstalliert werden. Auf einem neu eingerichteten Ubuntu-System erfolgt dies mit folgendem Befehl:

```
$ sudo apt-get install ncurses-dev gawk bison flex autoconf automake
```

Im WL-500 befindet sich ein Broadcom-Prozessor vom Typ 4704; weiterhin soll das Zielsystem zudem mit Kernelversion 2.6 arbeiten, wodurch sich insbesondere bei der Verwendung von USB-Komponenten und bei der Treiberprogrammierung Vorteile gegenüber Kernelversion 2.4 ergeben. Eine kleine Einschränkung hierbei ist, dass für die Kernelversion 2.6 momentan keine WLAN-Unterstützung für die eingebaute WLAN-MiniPCI-Karte verfügbar ist (vgl. Abschnitt 2.3).

Alternativ kann die bereits erwähnte Atheros-MiniPCI-Karte eingesetzt oder alternativ auf Kernel 2.4 ausgewichen werden. Dann ist allerdings die Verwendung von Treibern für den 2.6er Kernel (IO-Warrior, vgl. Abschnitt 3.6) nicht möglich. Eine Auswahl des Zielsystems *Broadcom BCM947xx/953xx [2.6]* wird deshalb empfohlen. Nun kann die Auswahl zusätzlicher Pakete erfolgen (z. B. *stty* für eine einfache Konfiguration der seriellen Schnittstelle). Diese können

<sup>3</sup> Bspw. eingesetzt in der Linksys NSLU2.



mit [\*] direkt in den Kernel eingebunden oder über die Option [M] als Modul gebaut werden, müssen dann allerdings vor der Verwendung installiert werden (vgl. Abschnitt 3.5). Um den Kernel möglichst schlank zu halten, sollte im Regelfall die zweite Möglichkeit angewandt werden.

Die Gesamtkonfiguration könnte z. B. folgendermaßen aussehen:

- Target System / [X] Broadcom BCM947xx/953xx [2.6]
- Base system / Busybox Configuration / Coreutils / [M] stty

Weiterhin besteht die Möglichkeit, Konfigurationsdateien zu speichern bzw. zu laden. Eine passende Konfiguration für die OpenWrt-Version 7.09 ist in den Quelltext-Dateien unter `src/openwrt/asus-wl-500.config` enthalten.

**Hinweis:** Für den Linksys WRT54G kann die gleiche Datei verwendet werden. Für eine Linksys NSLU2 mit ARM-Architektur ist abweichend davon die Datei `src/openwrt/nslu2.config` zu verwenden.

Nach erfolgter Konfiguration kann nun an der Kommandozeile der Übersetzungsprozess gestartet werden:

```
$ make
```

Das OpenWrt-Verzeichnis benötigt nach dem Übersetzen mindestens 2,5 Gigabyte Speicherplatz. Der Zusatz `V=99` liefert detaillierte Angaben während des Durchlaufs, die besonders bei der Erstellung eigener Module hilfreich sind (siehe Abschnitt 3.5).

**Hinweis:** Für das Zielsystem ARM ist die lizenzierte Intel-Bibliothek `IPL_ixp400NpeLibrary-2.4` notwendig. Diese ist nicht Open Source und kann entsprechend nur vorcompiliert bezogen werden.<sup>4</sup> Nach der Zustimmung zur *Intel Public License* und dem Herunterladen und Ablegen in `<openwrt-dir>/dl/` kann fortgefahren werden.

Nach erfolgreich durchlaufenem Übersetzungsprozess wurden vom Build-System zusätzliche Verzeichnisse angelegt. Die wichtigsten lauten wie folgt:

- `<openwrt-dir>/bin` – Enthält Flash-Images mit Basissystem und fertige Pakete im `.ipk`-Format.
- `<openwrt-dir>/build_mipsel` – Enthält zusätzliche, für das Zielsystem gebaute Anwendungen.
- `<openwrt-dir>/staging_dir_mipsel/bin` – Enthält die fertigen Cross Compiler.

---

<sup>4</sup> Diese Bibliothek ist im Download-Bereich der Intel-Website unter <http://downloadcenter.intel.com/> bei einer Suche nach „ixp400“ zu finden.

In dem Verzeichnis `<openwrt-dir>/bin` sollte ab jetzt eine Datei `openwrt-brcm47xx-2.6-squashfs.trx` vorhanden sein, welche in den Flash-Speicher des Routers übertragen werden kann. Für andere Geräte mit gleicher Architektur wie bspw. den Linksys WRT54g ist die Datei `openwrt-wrt54g-2.6-squashfs.bin` zu verwenden. Die Dateien mit Endung `.bin` stellen im Grunde `.trx`-Dateien mit Zusatzinformationen dar, um mit den Werkzeugen des Herstellers kompatibel zu bleiben. So lassen sich Firmware-Images alternativ auch über die Website des Routers flashen.

### 3.2.1 Aufspielen des Flash-Images

Bevor eine Binärdatei in den Flash-Speicher des Routers übertragen werden kann, muss dieser zunächst in den Diagnosemodus versetzt werden.

**Hinweis:** Dies ist nur für den WL-500 und die NSLU2 notwendig. Der WRT54 ist direkt nach dem Anlegen einer Spannung für kurze Zeit zur Aufnahme eines neuen Flash-Images bereit. Sollte dies nicht möglich sein, so hilft hier der Diagnosemodus weiter. Im Falle einer NSLU2 wird das Firmware-Image nicht mit `tftp`, sondern mit dem Programm `upslug2` aufgespielt. Dieses Programm ist als Debian-Paket verfügbar und erkennt eine an der Netzwerkschnittstelle des Host-Rechners angeschlossene NSLU2 automatisch, eine spezielle Konfiguration des PCs ist nicht notwendig.

Die Aktivierung des Diagnosemodus geschieht für alle Geräte unterschiedlich und ist für den WL-500, WRT54G und NSLU2 in Anhang A.5 beschrieben. Wurde dieser Schritt (für WL-500 oder NSLU2) durchgeführt, so lässt sich das Firmware-Image mittels `tftp` oder der neueren Version `atftp` (gegebenfalls über `sudo apt-get install tftp` bzw. `atftp` nachzuinstallieren) in den Flash-Speicher des WL-500 übertragen. Dazu werden im Verzeichnis `<openwrt-dir>/bin` folgende Befehle ausgeführt:

```
$ tftp 192.168.1.1
$ tftp> binary
$ tftp> trace
$ tftp> put openwrt-brcm47xx-2.6-squashfs.trx
```

**Wichtig:** Nachdem der Upload beendet ist, sollte mindestens fünf Minuten gewartet werden, da die Firmware zuerst in den Hauptspeicher geladen und von dort geflasht wird. Danach wird der Router automatisch neu booten, was eine Weile dauern kann. Einige Minuten nach dem Neustart sollte ein `ping 192.168.1.1` möglich sein, dies ist die Standard-IP nach der Übertragung der OpenWrt-Firmware. In einzelnen Fällen startet sich der Router nicht automatisch selbst von Neuem, es kann dann aber auch nach fünf Minuten manuell ein Neustart durchgeführt werden.

### 3.2.2 Der erste Einlog-Vorgang

Über den Befehl `telnet 192.168.1.1` kann sich der Anwender nun mit dem Router verbinden, und folgendes Cocktail-Rezept sollte auf dem Bildschirm erscheinen:

```
BusyBox v1.4.2 (2008-06-27 23:47:53 CEST) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

```

|-----|
|  -    | |  -    | |  -    | |  -    | |  -    | |  -    |
|-----|
|  | W I R E L E S S   F R E E D O M
|-----|
KAMIKAZE (7.09)
* 10 oz Vodka           Shake well with ice and strain
* 10 oz Triple sec      mixture into 10 shot glasses.
* 10 oz lime juice      Salute!
|-----|
root@OpenWrt:~#
```

Für das weitere Vorgehen bietet SSH eine sichere Alternative zu der unverschlüsselten Telnet-Verbindung. Zur Aktivierung einer SSH-Verbindung muss zunächst durch den folgenden Aufruf ein Passwort gesetzt werden:

```
$ passwd
```

Danach sollte sich der Anwender sofort mit `exit` wieder abmelden. Nun ist das Gerät via `ssh root@192.168.1.1` erreichbar; weiterhin ist Telnet von nun an deaktiviert. Für die weitere Installation wird eine Internetverbindung für den WL-500 benötigt. Dieser muss zunächst für einen Betrieb als Client im lokalen Hausnetz konfiguriert werden. Im Folgenden wird davon ausgegangen, dass ein weiterer Router existiert, welcher eine Internetverbindung besitzt und dessen IP bekannt ist.

Der WL-500 wird im einfachsten Fall mit einer statischen IP versehen. Der Anwender loggt sich hierfür ein und bearbeitet die Netzwerkeinstellungen in `/etc/config/network`. Bei einem Internet-Router mit IP 192.168.1.1 sollten die Einstellungen folgendermaßen aussehen:

```
#### LAN configuration
config interface lan
    option type        bridge
    option ifname      "eth0.0"
    option proto        static
    option ipaddr      <wl500-ip>
    option netmask      255.255.255.0
    option gateway      192.168.1.1
    option dns          192.168.1.1
```

Der einfache Editor `vi` ist auf fast jedem Linux-System vorhanden und kann entsprechend auch für das Editieren solcher Konfigurationsdateien verwendet werden. Hinweise zu der etwas gewöhnungsbedürftigen Bedienung finden sich in Anhang A.2.2.

Grundsätzlich ist die Vergabe einer statischen IP der sicherere Weg, da hiermit der WL-500 im Notfall auch direkt ohne DHCP-Server am Entwicklungs-PC betrieben werden kann. Wahlweise kann aber auch eine dynamische IP über DHCP bezogen werden. Für den zweiten Fall muss in der DHCP-Tabelle des Haus-Routers der MAC-Adresse des WL-500 eine bestimmte IP zugewiesen werden. Die MAC-Adresse ist auf dem Boden des Gerätes abgedruckt, kann aber auch über folgenden Befehl ermittelt werden:

```
$ ifconfig
```

Für eine DHCP-Anbindung müssen die Netzwerkeinstellungen in `/etc/config/network` wie folgt konfiguriert werden:

```
### LAN configuration
config interface lan
    option type        bridge
    option ifname      "eth0.0"
    option proto        dhcp
```

Auch auf dem WL-500 läuft standardmäßig ein DHCP-Server. Für den Betrieb als Client im Hausnetz wird dieser nicht benötigt und kann abgeschaltet werden:

```
$ /etc/init.d/dnsmasq disable
```

Die Konfiguration des WL-500 für den Anschluss am Heimnetz ist nun abgeschlossen, und das Gerät sollte unter der eingestellten IP-Adresse zu finden sein. Im nächsten Schritt wird der Host-PC auf die ursprünglichen Einstellungen zurückgesetzt, und es wird per SSH ein erneuter Einlog-Vorgang zum WL-500 gestartet. Nun sollte eine Verbindung zum Internet vorhanden sein – ein Test kann aussehen wie folgt:

```
$ ping google.de
```

In einem weiteren Schritt kann nun die Web-Oberfläche installiert werden, welche bei OpenWrt nicht enthalten ist. Sie wurde in das separate Projekt X-Wrt ausgelagert [X-Wrt 08]. Hierzu werden zunächst die ipkg-Paketinformationen<sup>5</sup> auf den neuesten Stand gebracht. Anschließend wird das Paket für die Web-Oberfläche installiert:

```
$ ipkg update
$ ipkg install http://downloads.x-wrt.org/xwrt/kamikaze/7.09/brcm47xx-2.6/webif_latest.ipk
```

Je nach Distribution und Gerät ist ein anderer Link zu verwenden; auf der Download-Seite des Projektes [X-Wrt 08] können die entsprechenden Pakete nachgeschaut werden. Die WL-500-Website ist nun unter voreingestellter

<sup>5</sup> ipkg steht für *Itsy Package Management System* [ipkg 08], ein ressourcenschonendes Paketsystem für Microrechner, Handhelds und Embedded-Systeme. Die Kontrollprogramme sind auf das Notwendigste beschränkt, und auch die .ipk-Pakete sind feingranular strukturiert und entsprechend schlank gehalten.

IP über einen Webbrowser zugänglich. Benutzernamen und Passwort sind die gleichen wie für den SSH-Login. Über die Web-Oberfläche können Statusinformationen abgefragt, reguläre Router-Einstellungen vorgenommen und ipkg-Pakete verwaltet werden.

### 3.3 Schnelleres Einloggen mit SSH-Keys

Bevor nun mit der Software-Entwicklung begonnen werden kann, stehen noch einige Vorbereitungen zur Erleichterung der späteren Arbeit an. Die Entwicklung auf dem Host-System und das Testen auf dem WL-500 bringen eine Vielzahl von SSH-Verbindungen und SCP-Kopiervorgängen (Secure Copy) mit sich. Der Anwender kann sich hierbei die häufige Eingabe des Passwortes ersparen, indem er SSH-Schlüssel verwendet.

Die Vorgehensweise unterscheidet sich etwas von jener bei Debian-basierten Systemen (siehe Anhang A.3.2) und ist aus diesem Grunde hier explizit dokumentiert. Für ein Einloggen ohne Passworтеingabe muss der öffentliche Schlüssel des Host-PCs auf dem WL-500 liegen. Falls auf dem Host noch kein Schlüsselpaar vom Typ `dsa` vorhanden ist, so kann dieses mit folgenden Befehlen erzeugt werden:

```
$ cd ~/.ssh
$ ssh-keygen -t dsa -f id_dsa
```

Nun wird der öffentliche Schlüssel vom Host aus in das Verzeichnis `/tmp` des WL-500 kopiert (das Verzeichnis liegt im flüchtigen Speicher):

```
$ scp ~/.ssh/id_dsa.pub root@wl500-ip>:/tmp
```

Dort eingeloggt muss der Schlüssel der Datei `authorized_keys` hinzugefügt werden, dies geschieht durch eine Weiterleitung der `cat`-Ausgabe in die Ziel-datei. In `authorized_keys` sind die öffentlichen Schlüssel aller Partner hinterlegt, die sich direkt am WL-500 anmelden dürfen. Weiterhin soll nur der Besitzer Lese- und Schreibrechte auf diese Datei erhalten (vgl. `chmod` in Abschnitt A.4.4):

```
$ cd /etc/dropbear
$ cat /tmp/id_dsa.pub >> authorized_keys
$ chmod 0600 authorized_keys
```

Im Unterschied zu Mehrbenutzersystemen wie Debian liegen das Schlüsselpaar und die Autorisierungsinformationen nicht im Home-Verzeichnis, sondern global in `/etc/dropbear`, dem Konfigurationsverzeichnis des SSH-Server- und Client-Programms. Wer Wert auf zusätzliche Sicherheit legt, der kann in der Datei `etc/config/dropbear` die Möglichkeit des Passwort-Logins deaktivieren oder auch gegebenenfalls den SSH-Port wechseln:

```
# option PasswordAuth 'on'
option PasswordAuto 'off'
# option Port '22'
option Port '9222'
```

**Wichtig:** Vorher sollte der Anwender den automatischen Log-in getestet haben, da er sich sonst mit dieser Umstellung u. U. aus dem System aussperrt. In einem solchen Fall bleibt als Ausweg nur das Rücksetzen der Einstellungen über die Web-Oberfläche.

Auch wenn die beschriebene Kommunikationsrichtung sicher der Regelfall ist, so kann es notwendig sein, den anderen Weg zu gehen, um beispielsweise vom WL-500 automatisch beim Neustart eine Datei von einem Server zu holen. In `/etc/dropbear` liegen dazu bereits zwei private Schlüssel `dropbear_rsa_host_key` und `dropbear_dss_host_key`, im folgenden interessiert lediglich der RSA-Schlüssel, der im Gegensatz zu DSS nicht nur signiert, sondern auch verschlüsselt. Der zugehörige öffentliche Schlüssel kann mit nachfolgendem Befehl erzeugt werden. Anschließend wird dieser auf jenes Host-System, auf welchem ein automatischer Log-in erfolgen soll, übertragen:

```
$ dropbearkey -y -f /etc/dropbear/dropbear_rsa_host_key | grep ssh-rsa > /
etc/dropbear/dropbear_rsa_host_key.pub
$ cp /etc/dropbear/dropbear_rsa_host_key.pub <user>@<host-ip>:
```

Nach der Anmeldung am Host-System kann der öffentliche WL-500-Schlüssel zu den bereits autorisierten Schlüsseln im Home-Verzeichnis des Benutzers hinzugefügt werden:

```
$ cat ~/.dropbear_rsa_host_key.pub >> ~/.ssh/authorized_keys
```

Vom WL-500 aus ist der Host-Rechner nun unter Angabe des öffentlichen Schlüssels mit der Option `-i` folgendermaßen per SSH oder SCP erreichbar:

```
$ ssh -i /etc/dropbear/dropbear_rsa_host_key <user>@<host-ip>
```

## 3.4 Software-Entwicklung für OpenWrt

Nachdem nun mittels OpenWrt BS die gesamte Toolchain mitsamt Cross Compilern erstellt wurde, ist die Verwendung denkbar einfach. Die Cross Compiler liegen im Verzeichnis `<openwrt-dir>/staging_dir_mipsel/bin` und tragen die Namen `mipsel-linux-uclibc-<...>`. Durch Hinzufügen der folgenden Zeile zu `~/.bashrc` sind die Cross Compiler nach einem Neustart der Shell automatisch verfügbar:

```
export PATH=<openwrt-dir>/staging_dir_mipsel/bin/:$PATH
```

Die Makefile-Dateien in den mitgelieferten Beispielen sind so aufgebaut, dass neben Standard-Compilern auch eben jene Cross Compiler als Alternative angegeben werden können. Im Embedded-Linux-Quellverzeichnis sind diese Einstellungen in der Datei `src/Makefile.config` enthalten:

```
CXX_STD    = g++
CXX_CROSS  = mipsel-linux-uclibc-g++

CC_STD     = gcc
CC_CROSS   = mipsel-linux-uclibc-gcc
```

Weiterhin können für Compiler und Linker je nach Compiler-Wahl verschiedene Flags angegeben werden. Jede Make-Datei im mitgelieferten Quelltext bindet `Makefile.config` ein, sodass der Aufruf `make` den regulären Compiler verwendet, während mit der Option `make CROSS=1` der Cross Compiler zum Einsatz kommt. Die gewählten Compiler für C und C++ werden, wie allgemein üblich, in den Umgebungsvariablen `CC` und `CXX` hinterlegt<sup>6</sup>. Die Beispiele `examples/sockets/udp_basic_sender` und `examples/sockets/udp_basic_receiver` sollen dies veranschaulichen und eine Zeichenkette von einem x86-basierten System auf den WL-500 schicken. Das Sendeprogramm wird zunächst für die Host-Plattform übersetzt:

```
$ cd <openwrt-dir>/examples/udp/udp_basic_sender
$ make
```

Ein Aufruf von `file` zeigt, dass es sich um eine ausführbare Datei für Intel-80386-Prozessoren handelt:

```
$ file sender
$ sender: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
          GNU/Linux 2.6.8, dynamically linked (uses shared libs), not stripped
```

Durch Wechseln in das Verzeichnis des Empfänger-Programms und Übersetzung mit dem Cross Compiler entsteht dann eine Datei `receiver`, die für die MIPSel-Architektur übersetzt wurde:

```
$ cd <openwrt-dir>/examples/sockets/udp_basic_receiver
$ make CROSS=1
```

Bei auftretenden Problemen sind die Einstellungen in `src/Makefile.config` zu prüfen. Ein Test mit `file` bestätigt wieder das Ergebnis:

```
$ file receiver
$ receiver: ELF 32-bit LSB executable, MIPS, version 1 (SYSV), dynamically
           linked (uses shared libs), not stripped
```

Jetzt kann die Anwendung auf den Router kopiert und dort gestartet werden:

<sup>6</sup> Die Make-Dateien sind so aufgebaut, dass die Objektdateien, auch wenn sie gemeinsam genutzten Quellen entstammen, nicht an zentraler Stelle, sondern immer im lokalen Verzeichnis abgelegt werden. Das erhöht zwar den Compilieraufwand, spart allerdings beim Wechseln vom g++ zum Cross Compiler die Linkerfehler, welche aus Objektdateien mit verschiedener Architektur resultieren.

```
$ scp receiver root@wl500-ip:
-> Wechsel auf WL-500
$ ./receiver 64000
```

Der Zusatz 64000 gibt den Port an, auf welchem die UDP-Daten empfangen werden sollen. U. U. führt nun die fehlende C++-Standardbibliothek zu einer Fehlermeldung beim Starten von **receiver** auf dem WL-500:

```
$ ./receiver: can't load library 'libstdc++.so.6'
```

Die Bibliothek kann dann mit **ipkg** einfach nachinstalliert werden:

```
$ ipkg install libstdcpp
```

Nun sollte sich die Anwendung **receiver** mit Angabe der Portnummer starten lassen. Im Anschluss kann vom Host-Rechner aus eine Zeichenkette eingegeben und gesendet werden, die auf dem WL-500 empfangen und ausgegeben wird:

```
$ ./sender <wl500-ip> 64000
```

Die Cross Compiler können natürlich auch in einer integrierten Entwicklungsumgebung wie beispielsweise Eclipse verwendet werden [Eclipse 08]. Hierzu wird in den Projekteinstellungen unter *C/C++ Build* eine weitere Konfiguration angelegt, und es werden bei den dortigen Einstellungen *Settings* die Einträge für Compiler, Linker und Assembler in Aufrufe des Cross Compilers geändert.

## 3.5 Erstellung eigener OpenWrt-Module

Die im vorigen Abschnitt beschriebene Methode der Software-Entwicklung für einen OpenWrt-basierten Router ist für das schnelle Testen einer Anwendung völlig ausreichend. Ist das Projekt aber schon weiter gediehen, so sollte u. U. ein eigenes OpenWrt-Paket erstellt werden. Hiermit lassen sich dann alle Vorteile der **ipkg**-Paketverwaltung wie z. B. Versionsverwaltung und Funktionen zur Instandhaltung nutzen.

Einmal lässt sich damit Ordnung in das System bringen. Weiterhin können so mehrere OpenWrt-Geräte leicht auf einem einheitlichen Stand gehalten werden. Und abschließend können damit auch im Sinne des Open-Source-Gedankens die eigenen Entwicklungen anderen OpenWrt-Benutzern einfach zugänglich gemacht werden.

Dreh- und Angelpunkt bei der Integration in das OpenWrt-Build-System ist die Beschreibung des eigenen Paketes in Form einer Datei `<openwrt-dir>/package/<pack_name>/Makefile`. Falsche Einträge an dieser



Stelle können zu schwer nachvollziehbaren Fehlern führen. Anhand des Quelltextes in `<embedded-linux-dir>/src/openwrt/helloworld.pck/` soll mithilfe eines Hello-World-Beispiels der Aufbau einer solchen Paketbeschreibung erklärt werden.

Für die Erstellung eines neuen Paketes muss zunächst ein Paketverzeichnis angelegt werden:

```
$ mkdir <openwrt-dir>/package/helloworld
```

Dort wird die Projektbeschreibung `Makefile`<sup>7</sup> platziert. Diese Datei enthält eine Beschreibung des Paketes, die Angabe der Quelltext-Dateien, Informationen zum Build-Verzeichnis und ein Installationsziel für fertige Pakete. Besondere Aufmerksamkeit ist hinsichtlich des korrekten Setzens von Leerzeichen und Tabstopps notwendig; Zum einen sind für Einrückungen statt Leerzeichen nur Tabstopps erlaubt, zum anderen dürfen auch an Zeilenenden keine Leerzeichen vorkommen.

Neben der Datei `Makefile` werden im Paketverzeichnis auch die Quellen benötigt und entsprechend an diesen Ort kopiert:

```
$ cp <embedded-linux-dir>/src/openwrt/helloworld\_pck/Makefile <openwrt-dir>/package/helloworld/
$ cp -r <embedded-linux-dir>/src/openwrt/helloworld\_pck/helloworld-1.0.0 <openwrt-dir>/package/helloworld/
```

Die zeilenweise Betrachtung von `Makefile` zeigt, dass nach dem Setzen OpenWrt-interner Variablen in `rules.mk` die Angabe von Paket, Versionsnummer und Release erfolgt. Aus dieser Beschreibung werden später Dateinamen für die Quellen nach der Konvention `$(PKG_NAME)-$(PKG_VERSION)` zusammengesetzt.

```
include $(TOPDIR)/rules.mk

# Name and release number of this package
PKG_NAME:=helloworld
PKG_VERSION:=1.0.0
PKG_RELEASE:=1
```

Durch den folgenden Ausdruck wird angegeben, wo das Paket erstellt wird:

```
# Build-Directory
PKG_BUILD_DIR:=$(BUILD_DIR)/$(PKG_NAME)-$(PKG_VERSION)
```

Die Variable `$(BUILD_DIR)` ist standardmäßig auf das Verzeichnis `<openwrt-dir>/build_mipsel` gesetzt. Die darauf folgenden Zeilen dienen der Paketbeschreibung und der Einordnung in die OpenWrt-Paketstruktur. Unter der angegebenen Kategorie taucht das Paket in der Konfigurationsliste (`make menuconfig`) auf und kann entsprechend dort eingebunden werden.

<sup>7</sup> `<embedded-linux-dir>/src/openwrt/helloworld.pck/Makefile.`

```
# Set package category under which it appears in configuration
define Package/helloworld
    SECTION:=utils
    CATEGORY:=MyPackages
    TITLE:=Hello World
endef

# Set package description
define Package/helloworld/description
    This is just a test package
endef
```

Die Angabe der für die Vorbereitung auf den eigentlichen Build-Prozess erforderlichen Aktionen wird über **Build/Prepare** spezifiziert. Im vorliegenden Fall wird das Build-Verzeichnis erstellt (das zu diesem Zeitpunkt noch nicht vorhanden ist) und die im Paketverzeichnis liegenden Quellen werden dorthin kopiert:

```
# Copy local source files from package into build directory to prepare
    build process
define Build/Prepare
    mkdir -p $(PKG_BUILD_DIR)
    $(CP) ./$(PKG_NAME)-$(PKG_VERSION)/ * $(PKG_BUILD_DIR)/
endef
```

Diese Vorgehensweise ist weniger üblich, normalerweise werden die Quellen für die Pakete nicht mitgeliefert, sondern in Form einer Datei **name.tar.gz** aus dem Internet geladen.

Im vorliegenden Fall soll aber die Variante mit lokalen Quellen verwendet werden, um auf den sonst notwendigen Webserver verzichten zu können. Eine Paketbeschreibung des gleichen Beispiels, aber mit Bezug der Quelltext-Dateien von einem HTTP-Server, befindet sich als Datei **Makefile.http** im mitgelieferten Quellverzeichnis des Paketes und kann alternativ verwendet werden (nach einer Umbenennung in **Makefile**).

Über die Prämisse **Package/helloworld/install** wird festgelegt, was bei einer Installation des Paketes genau geschehen soll. **\$(INSTALL\_BIN)** enthält die Anweisung, um die Binärdateien aus dem Build-Verzeichnis nach **\$(1)/bin/** zu kopieren. **\$(1)** stellt dabei das Wurzelverzeichnis **/** auf dem Router dar. Mit **\$(INSTALL\_DIR)** wird zuvor sichergestellt, dass dieses Verzeichnis existiert. Ansonsten wird es angelegt:

```
# Specify where to install the binaries
define Package/helloworld/install
    $(INSTALL_DIR) $(1)/bin
    $(INSTALL_BIN) $(PKG_BUILD_DIR)/helloworld $(1)/bin/
endef
```

Die folgende Zeile führt die notwendigen Befehle aus, um das Paket mit den vorgenommenen Einstellungen zu übersetzen. Der Name muss mit dem Verzeichnisnamen des Paketes übereinstimmen:

```
# Compile package
$(eval $(call BuildPackage,helloworld))
```

Wird die Variante mit Quellenbezug aus dem Internet gewählt, wie in **Makefile.http** hinterlegt (umbenennen nicht vergessen), so sind in der Paketbeschreibung weitere Zeilen hinsichtlich der notwendigen Informationen zu Quelle und Prüfsumme anzugeben:

```
PKG_SOURCE_URL:=http://www.praxisbuch.net/embedded-linux/
PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.gz
PKG_CAT:=zcat
PKG_MD5SUM:=a53e15cfb4e0955a3c4b64429fc5d1d9
```

Die Angabe der MD5-Prüfsumme ist optional. Ist aber die Prüfsumme vorhanden, so muss sie übereinstimmen, sonst wird die Archivdatei nach dem Download verworfen<sup>8</sup>. Die Quellen werden nach dem Download im Verzeichnis `<openwrt-dir>/dl/` abgelegt. Das OpenWrt BS entpackt die Quellen automatisch in das Verzeichnis `$(PKG_BUILD_DIR)`. Auf die Definition von weiteren Anweisungen in **Build/Prepare** kann in diesem Fall verzichtet werden.

Wird nun die Konfigurationsumgebung mit dem folgenden Aufruf gestartet, so sollte in der Kategorie **MyPackages** das Paket **helloworld** mitsamt Beschreibung erscheinen:

```
$ make menuconfig
```

Das Paket kann nun wahlweise in den Kernel übernommen oder nur als Modul übersetzt werden. Die zweite Möglichkeit reicht aus, sodass nach Abspeichern und Übersetzen mit **make** im Verzeichnis `<openwrt-dir>/bin/packages/` eine neue Paketdatei **helloworld\_1.0.0-1\_mipsel.ipk** liegen sollte. Ein einzelnes Paket lässt sich übrigens auch direkt mit **make package/<pack\_name>-compile** bauen. Wird diese Datei auf den Router kopiert, so kann das Paket dort folgendermaßen installiert werden:

```
$ ipkg install helloworld_1.0.0-1_mipsel.ipk
```

Anschließend liegt die ausführbare Anwendung global verfügbar im Verzeichnis `/bin/` und der WL-500 schickt nach einem Aufruf Grüße in die Welt:

```
$ ./helloworld
$ Hello World!
```

Weitere Tipps zur Erstellung eigener Pakete sind in der *Einführung in das OpenWrt BS* im Forum verfügbar [OpenWrt BS Intro 08].

<sup>8</sup> Die MD5-Prüfsumme einer Datei kann in der Kommandozeile durch den Aufruf von `md5sum <dateiname>` angezeigt werden.

## 3.6 IO-Warrior-Erweiterung und Kernelmodule unter OpenWrt

Die IO-Warrior-Bausteine der Firma Code Mercenaries werden ausführlich in Abschnitt 7.5 vorgestellt, und dort wird auch die Installation unter Debian erläutert. An dieser Stelle soll lediglich auf die Besonderheiten bei einem Betrieb unter OpenWrt eingegangen werden.

Zum Betrieb eines IO-Warriors wird ein Kernel-Modul benötigt, welches Routinen zur Kommunikation mit dem IO-Warrior-Baustein bereitstellt und auch weitere Kernel-Module zum USB-Zugriff verwendet. Die Erstellung eines Kernel-Moduls für OpenWrt erfolgt ähnlich der Erzeugung eigener Pakete (vgl. Abschnitt 3.5), es müssen allerdings noch einige weitere Details beachtet werden.

Zunächst wird für das neue Paket ein Verzeichnis im Paketordner angelegt:

```
$ mkdir <openwrt-dir>/package/iowarrior
```

Wie jedes OpenWrt-Paket, so müssen auch Kernel-Module über eine Beschreibung in Form einer Datei **Makefile** in das OpenWrt BS integriert werden. Für das Paket **kmod-iowarrior** sieht diese folgendermaßen aus<sup>9</sup>:

```
include $(TOPDIR)/rules.mk
include $(INCLUDE_DIR)/kernel.mk

# Name and release number of this package
PKG_NAME:=kmod-iowarrior
PKG_RELEASE:=1

# Build-Directory
PKG_BUILD_DIR:=$(KERNEL_BUILD_DIR)/$(PKG_NAME)

include $(INCLUDE_DIR)/package.mk

# Set package category under which it appears in configuration,
# dependancies, description and target file
define KernelPackage/iowarrior
    SUBMENU:=Other modules
    DEPENDS:=@PACKAGE_kmod-usb-core
    TITLE:=IOwarrior driver
    DESCRIPTION:=\
        This package contains the driver for IOWarriors 24, 40 and 56.
    VERSION:=$(LINUX_VERSION)-$(BOARD)-$(PKG_RELEASE)
    FILES:= \
        $(PKG_BUILD_DIR)/iowarrior.$(LINUX_KMOD_SUFFIX)
    AUTOLOAD:=$(call AutoLoad,80,usb-core)
endef

# Copy local source files from package into build directory to prepare
# build process
define Build/Prepare
    mkdir -p $(PKG_BUILD_DIR)
    $(CP) ./src/* $(PKG_BUILD_DIR)/
endef
```

<sup>9</sup> Verfügbar unter `<embedded-linux-dir>/src/openwrt/kmod-warrior/Makefile`.

```
# Commands for the compile process
define Build/Compile
    $(MAKE) -C "$(LINUX_DIR)" \
        CROSS_COMPILE="$(TARGET_CROSS)" \
        ARCH="$(LINUX_KARCH)" \
        SUBDIRS="$(PKG_BUILD_DIR)" \
        EXTRA_CFLAGS="$(BUILD_FLAGS)" \
        modules
endef

# Specify what to do in the installation process
# $(INSTALL_BIN) copies udev rules
define KernelPackage/iowarrior/install
    $(INSTALL_DIR) $(1)/etc/udev/rules.d/
    $(INSTALL_BIN) ./src/10-iowarrior.rules $(1)/etc/udev/rules.d/
endef

# Compile package
$(eval $(call KernelPackage,iowarrior))
```

Besonders relevant ist der Zusatz `KernelPackage` am Dateiende. Hiermit wird angezeigt, dass es sich beim vorliegenden Paket nicht um ein reguläres Modul, sondern um ein Kernel-Modul handelt. Es muss entsprechend auch anders übersetzt werden.

Weiterhin unterscheiden sich Build-Verzeichnis und Paketbeschreibung von jenen eines regulären Paketes.

Nach dem Kopieren der Dateibeschreibung und des Quellcodes und einem anschließenden Aufruf des Konfigurationseditors sollte das eigene Modul `kmod-iowarrior` in der Rubrik `Kernel modules->Other modules` auftauchen:

```
$ cp -r <embedded-linux-dir>/src/openwrt/kmod-iowarrior/* <openwrt-dir>/
package/iowarrior/
$ make menuconfig
```

Um das Flash-Image nicht neu aufspielen zu müssen, wird das Modul nicht per `<*>` in den Kernel integriert, sondern lediglich mit `<M>` als Modul gebaut.

Nach dem Beenden und Übersetzen mittels `make` liegt nach einigen Minuten im Verzeichnis `<openwrt-dir>/bin/packages/` eine Datei `kmod-iowarrior_2.6.22-brcm47xx-1_mipsel.ipk`. Diese ist auf den WL-500 zu übertragen und dort über folgenden Aufruf zu installieren:

```
$ ipkg install kmod-iowarrior_2.6.22-brcm47xx-1_mipsel.ipk
```

Durch die Installationsanweisungen sollte nun auch in `/etc/udev/rules.d/` ein Eintrag `10-iowarrior.rules` angelegt worden sein. In dieser Datei ist die Verbindung zwischen Kernel-Modul und Gerätenamen hinterlegt. Erkennt das IO-Warrior-Modul eine angeschlossene Platine, so werden neue Geräte automatisch als `/dev/iowarrior<num>` erzeugt. Da das Kernel-Modul auf USB-Basisfunktionen angewiesen ist, werden auch die Pakete `usbcore` und `uhci_hcd` benötigt. Abschließend stellt das Paket `usbutils` wichtige Befehle wie z. B.

**lsusb** zur Verfügung, die im Umgang mit USB-Geräten häufiger Anwendung finden:<sup>10</sup>

```
$ ipkg install kmod-usb-core
$ ipkg install kmod-usb-uhci
$ ipkg install usbutils
```

Nach einem Neustart kann nun das Modul **iowarrior** geladen werden:

```
$ insmod iowarrior
```

Die Ausgabe von **lsmod** sollte das neue Modul unter den geladenen Modulen auflisten und dessen Benutzung des Moduls **usbcore** anzeigen. Der Befehl **dmesg**<sup>11</sup> zeigt den Nachrichtenpuffer des Kernels an. Dieser enthält als letzten Eintrag eine Nachricht des Moduls **usbcore**, welches mitteilt, dass gerade ein neuer Treiber registriert wurde. Sollten Probleme beim Laden der Module auftreten (hervorgerufen durch eine falsche Reihenfolge oder unterschiedliche Kernelversionen), so ist dies eine Möglichkeit, der Ursache auf den Grund zu gehen.

Nun ist es sinnvoll, das IO-Warrior-Modul beim Start automatisch zu laden. Eine saubere Vorgehensweise hierfür ist die Erstellung eines Skripts im Verzeichnis **/etc/init.d/**. Dazu ist eine Datei **/etc/init.d/iowarrior** anzulegen und folgender Inhalt einzugeben:

```
START=99

start () {
    insmod iowarrior
}
```

Die Startnummer gibt dabei die Reihenfolge des Skripts an, um die Aufrufe zu ordnen. Im vorliegenden Fall wird das Skript erst ganz am Ende aufgerufen. Eine Aktivierung legt das Skript in **/etc/rc.d/** ab, und beim nächsten Systemstart wird das Modul automatisch geladen:

```
$ /etc/init.d/iowarrior enable
```

Eine wichtige Anwendung fehlt noch, bevor die ersten Tests erfolgen können: Für die Erzeugung von Geräten wurden bereits Regeln hinterlegt. Das eigentliche Programm **udev** zur Geräteverwaltung muss jedoch noch über folgenden Befehl nachinstalliert werden:

```
$ ipkg install udev
```

Das Nachladen der Regeln ohne die Notwendigkeit eines Neustarts erfolgt durch den Befehl **udevcontrol**:

```
$ udevcontrol reload_rules
```

<sup>10</sup> Sollte **lsusb** trotz eines eingesteckten Adapters kein Gerät anzeigen, so hilft oftmals ein Aufruf von **mount -t usbfs none /proc/bus/usb**.

<sup>11</sup> Diagnostic Message.

Hiermit ist nun die Basis für einen ersten Testlauf geschaffen: Nach dem Einstecken der IO-Warrior-Platine zeigt ein Aufruf von `lsusb`, ob das neue USB-Gerät erkannt wurde. Mit etwas Glück wurden jetzt sogar zwei Geräte als `/dev/iowarrior0` bzw. `/dev/iowarrior1` erzeugt, die zur Kommunikation in den beiden Modi *Standard* und *Special* verwendet werden können.

Die mitgelieferten I<sup>2</sup>C-Beispiele in `<embedded-linux-dir>/examples/iic` können mit dem Zusatz `CROSS=1` cross-compiliert, und auf dem WL-500 ausgeführt werden:<sup>12</sup>

```
$ make CROSS=1
```

Das Beispiel `<embedded-linux-dir>/examples/iic/iic_iowarrior` zeigt die Verwendung der `iowarrior_i2c`-Routinen: Zuerst werden alle vorhandenen IO-Warrior-Geräte mit Seriennummern aufgelistet. Dann sucht das Programm nach einer Platine mit der Seriennummer `00001FEC`, um bei Erfolg einen daran angeschlossenen Temperatursensor vom Typ DS1631 mit Adresse `0x4f` auszu-lesen.

Für eine ausführliche Erklärung der `iowarrior_i2c`-Hilfsroutinen sei an dieser Stelle auf Abschnitt 8.3.4 verwiesen.

---

<sup>12</sup> Zu beachten ist dabei der passende Aufruf zur Erstellung des I<sup>2</sup>C-Bus-Objektes als *nativ* oder *IO-Warrior* – nähere Informationen hierzu werden in Kapitel 9 gegeben.

## Debian auf dem NAS-Gerät NSLU2

### 4.1 Einführung

Die NSLU2 von der Firma Linksys wurde im Jahre 2004 eingeführt und hat in der Linux-Community als günstige Hardware-Plattform rasch viele Fans gefunden. Dies führte zur Entwicklung zahlreicher Linux-Varianten wie *Unslung*, *SlugOS/LE* oder auch *Debian/NSLU2*. Bei der zuletzt genannten Debian-Variante handelt es sich um einen offiziellen Zweig der Debian-Distribution für die NSLU2-ARM-Architektur. Diese offizielle Aufnahme sorgt dafür, dass Debian/NSLU2 die volle Unterstützung der Debian-Community erfährt und die Anwender von den Paketen im Debian-Repository profitieren können. Damit stehen einerseits viele Programme zur Verfügung, um bspw. kostengünstig einen Netzwerkservers mit geringem Stromverbrauch aufzusetzen, andererseits können viele Anwender ihre Debian-Erfahrungen, die sie am PC gesammelt haben, direkt nutzen. Diese Vorteile haben dazu beigetragen, dass die Debian-Variante eine wesentlich höhere Verbreitung als die anderen genannten Distributionen gefunden hat.

Abschnitt 4.2 beschreibt die Installation von Debian/NSLU2. Im Anschluss werden in Abschnitt 4.3 die ersten Schritte nach einer Neuinstallation erklärt. Der Abschnitt 4.4 beschreibt die Software-Entwicklung für die NSLU2, Abschnitt 4.5 erläutert die Einrichtung eines Netzwerkdruckers mit CUPS. Weiterführende Hinweise werden im letzten Abschnitt 4.6 gegeben.

Die Kombination Debian + NSLU2 stellt ein vergleichsweise anwenderfreundliches System dar. Entsprechend fällt der Umfang dieses Kapitels auch etwas geringer aus – fast alle Debian-Anleitungen, die für den PC existieren, lassen sich auch auf Debian/NSLU2 anwenden.



## 4.2 Debian-Installation

Für die Installation sollte ein USB-Speicherstick mit mindestens zwei, besser vier Gigabyte Speicherplatz vorhanden sein. Es lohnt sich, hier etwas mehr Geld für schnellere Übertragungsraten von 20–30 MB/s zu investieren. Der im Vergleich zu *OpenWrt* oder *Puppy Linux* höhere Speicherplatzbedarf ist der Preis für ein umfassendes Debian-System. Wenn für eine spezielle Aufgabe besonders viel Speicherplatz erforderlich ist, so kann anstelle des USB-Sticks auch eine externe Platte verwendet werden.

Der in der NSLU2 eingesetzte Baustein vom Typ Intel IXP4xx wird vom Debian-Port offiziell unterstützt. Entsprechend existiert ein Debian-Installer<sup>1</sup>, um die Basis-Funktionalität zur Verfügung zu stellen und um den Anwender durch den Installationsprozess zu führen. Leider sind die Netzwerktreiber für den IXP4xx-Prozessor nicht Open-Source und deshalb aus Lizenzgründen nicht im Installer enthalten.

Bei der Verwendung des regulären Installers wäre deshalb ein zusätzlicher USB-Netzwerkadapter notwendig und während der Installation ein häufiger Wechsel zwischen Adapter und Ethernet-Anschluss der NSLU2 unabdingbar.<sup>2</sup>

Die zweite Variante ist wesentlich unkomplizierter: In einem inoffiziellen Debian-Installer<sup>3</sup> ist der Microcode für die Netzwerktreiber bereits enthalten. Bevor der Debian-Installer aufgespielt wird, sollte unbedingt über die Original-Web-Oberfläche sichergestellt werden, dass die (zukünftigen) Netzwerkeinstellungen korrekt und vollständig sind. Am sichersten ist es, hier eine statische IP-Adresse aus dem lokalen Hausnetz anzugeben, da die NSLU2 während der Installation unbedingt Netzzugang benötigt.<sup>4</sup> Neben IP, Subnetzmaske und Gateway (Router-IP) muss auch ein Nameserver eingetragen werden (üblicherweise übernimmt diese Aufgabe auch das Gateway). Nur wenn die Einstellungen vollständig sind, werden diese verwendet – ansonsten erfolgt eine DHCP-Anfrage.

Um den Installer in Form der Datei `di-nslu2.bin` in den Flash-Speicher der NSLU2 zu übertragen, existieren zwei Möglichkeiten:

---

<sup>1</sup> Vgl. <http://people.debian.org/~kmuto/d-i/images/daily/ixp4xx/netboot/>.

<sup>2</sup> Neben der Möglichkeit, Debian über einen Installer neu aufzusetzen, kann auch ein von Martin Michlmayr vorgefertigtes Paket auf den Speicherstick entpackt werden. Diese Lösung ist in <http://www.cyrius.com/debian/nslu2/unpack.html> beschrieben.

<sup>3</sup> Der Installer `di-nslu2.bin` kann von <http://www.slug-firmware.net/d-dls.php> nach dem Akzeptieren der Intel-Lizenzbedingungen heruntergeladen werden.

<sup>4</sup> Die NSLU2 besitzt im Auslieferungszustand die IP-Adresse 192.168.1.77. Um initial auf die Weboberfläche zugreifen, und Netzwerkeinstellungen anpassen zu können, ist der Host-PC ist für den direkten Anschluss mit einer entsprechenden IP-Adresse 192.168.1.xxx im Subnetz zu konfigurieren.



IP, unter welcher die NSLU2 nach dem Flashen erreichbar ist, hängt von den Voreinstellungen ab:

- Falls bereits eine Einstellung über die Linksys Web-Oberfläche vorgenommen wurde, so werden diese Einstellungen beibehalten. Sind die Einstellungen unvollständig, so bootet der Debian-Installer nicht. In diesem Fall muss die Original-Linksys-Firmware wieder aufgespielt werden.<sup>7</sup>
- Wurden die Originaleinstellungen aus dem Auslieferungszustand beibehalten, stellt die NSLU2 zunächst eine DHCP-Anfrage. Bleibt diese unbeantwortet, dann erfolgt der Start mit der Standard-Adresse 192.168.1.77.

Der Log-in auf der NSLU2 erfolgt vom Host-PC aus als Benutzer **installer** mit Passwort **install**:

```
$ ssh installer@<nslu2-ip>
```

Per Textmenü wird man nun durch den Debian-Installationsvorgang geführt. Die einzelnen Schritte sind selbsterklärend und fast identisch mit der regulären Debian-Installation auf einem PC. Am Ende des Vorganges wird der fertige Kernel in den Flash-Speicher geladen und das neue System gebootet.

Aufgrund der Einschränkung hinsichtlich Speicherplatz werden standardmäßig nicht alle Installer-Module geladen, sondern müssen manuell ausgewählt werden. Folgende Module sind für die Installation unbedingt erforderlich und werden durch Drücken der Leertaste hinzugefügt:

- **autopartkit**: Automatische Partitionierung der Festplatten,
- **ext3-modules-2.6.18-6-ixp4xx-di**: Unterstützung des EXT3-Dateisystems,
- **partman-auto**: Automatische Partitionierung von Speichermedien,
- **partman-ext3**: partman-Erweiterung für das EXT3-Dateisystem,
- **usb-storage-modules-2.6.18-6-ixp4xx-di**: Unterstützung von Speichermedien mit USB-Schnittstelle.

Über manuelle Partitionierung sollte mindestens eine Datenpartition vom Typ **ext3** und eine **swap**-Partition für virtuellen Speicher angelegt werden. Die **swap**-Partition muss mindestens 256 MB umfassen, da sonst während der Installation nicht ausreichend virtueller Speicher vorhanden ist und die SSH-Verbindung abbricht.<sup>8</sup> Eine geführte Partitionierung liefert erfahrungsgemäß eine zu geringe **swap**-Größe.

<sup>7</sup> Läuft bei der Installation etwas schief, so kann jederzeit die Original-Firmware mit **upslug2** auch ohne bekannte IP aufgespielt werden. Die Firmware ist auf <http://www.linksys.com> verfügbar (Suchbegriff: „nslu2 download“).

<sup>8</sup> Wenn bei der Formatierung Probleme auftreten oder wenn der USB-Stick nicht erkannt wird, so kann diese auch auf dem Host-PC mithilfe des Werkzeugs *GParted* erfolgen. Der Partitionsassistent kann damit zwar nicht übersprungen werden, man spart sich allerdings den nicht immer stabilen Formatierungsvorgang.

Bei der Frage nach zu installierender Software reicht es aus, lediglich das Standard-System auszuwählen. Weitere Dienste können nachträglich hinzugefügt werden. Auf eine Installation des *Desktop Environment* sollte in jedem Fall verzichtet werden. Das darin enthaltene, sehr umfangreiche X-Window-System würde die NSLU2 komplett überfordern. Die folgende Installation dauert knapp drei Stunden. Dann wird der neue Kernel in den Flash-Speicher übertragen und das System neu gestartet. Nach Beendigung der SSH-Verbindung dauert es einige Minuten, bevor ein Login via SSH als **root** oder als neu angelegter Benutzer möglich ist:

```
Linux localhost 2.6.18-4-ixp4xx #1 Tue Mar 27 18:01:56 BST 2007 armv5tel

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@localhost:~$
```

Wenn bei der Installation ein Fehler auftritt, so kann der Vorgang einfach wiederholt werden. Zuerst muss dann allerdings auf dem Host-PC die Schlüsseldatei `/home/<user>/.ssh/known-hosts` umbenannt bzw. der neue Schlüsseleintrag gelöscht werden.

## 4.3 Erste Schritte

Erfahrungsgemäß fehlen auf einem neuinstallierten System immer einige der notwendigen Basispakete. Diese sollten zu Beginn nachinstalliert werden. Nach einer Anmeldung als **root** ist es sinnvoll, zunächst die Uhrzeit korrekt zu setzen, um Fehlermeldungen bei der Installation neuer Pakete zu vermeiden:

```
$ ssh root@<nslu2-ip>
$ date -s "Oct 1 12:00:00 2008"
```

Nun kann die Aktualisierung der Paketinformationen und die anschließende Installation weiterer Pakete erfolgen:

```
$ apt-get update
$ apt-get install sudo ntpdate
```

Das Programm **sudo** erlaubt es, Benutzern temporäre **root**-Rechte zuzuweisen. Damit erübrigt sich ein Arbeiten als **root**, was stets vermieden werden sollte. Der bei der Installation angelegte Benutzer sollte deshalb in die Liste der **sudoers** aufgenommen werden (vgl. Abschnitt A.4.5 im Anhang). **ntpdate** dient dazu, die Systemzeit mit einem offiziellen Zeitserver im Internet abzugleichen und auf diese Weise zu aktualisieren.

Über folgenden Befehl wird die Systemzeit vom angegebenen Zeitserver<sup>9</sup> übernommen:

```
$ ntpdate time2.one4vision.de
```

Da sich geringe Zeitabweichungen aufsummieren und pro Tag bis zu 10 Sekunden betragen können, empfiehlt es sich, die Zeit regelmäßig abzugleichen. Dies kann automatisch über einen **cronjob** geschehen (vgl. Abschnitt A.4.6 im Anhang). Soll bspw. jede Nacht um 10 Minuten nach 1:00 Uhr die Zeit abgeglichen werden, so ist in der **crontab**-Datei des Benutzers **root** folgender Eintrag zu erstellen:

```
$ crontab -e
```

```
# m h dom mon dow   command
10 01 * * * /usr/sbin/ntpdate time2.one4vision.de > /dev/null
```

Eine Anmerkung zu Flash-Speichern: Wird bei der Installation ein USB-Speicherstick verwendet, so sollte sich der Anwender der limitierten Anzahl Schreibzyklen bewusst sein (vgl. auch Abschnitt 1.3.3, dort finden sich auch genauere Angaben zu der Zyklenzahl). Entsprechend sollte durch bestimmte Anpassungen die Zahl der Zugriffe reduziert werden.

Abschließend eine Empfehlung: Bei wiederholter SSH-Anmeldung kann die lästige Eingabe des Passwortes entfallen, wenn SSH-Schlüssel angelegt werden. Abschnitt A.3.2 beschreibt die notwendigen Schritte.

## 4.4 Software-Entwicklung für die NSLU2

Die Entwicklung eigener Software gestaltet sich auf der NSLU2 vergleichsweise einfach. Im Grunde müssen lediglich die notwendigen Bibliotheken und Programme wie Compiler und **make** installiert werden. Das Paket **build-essential** enthält die komplette Sammlung:

```
$ sudo apt-get install build-essential
```

Jetzt können die mitgelieferten Beispiele direkt auf der NSLU2 übersetzt werden. Nach der Übersetzung des **helloworld**-Beispiels in `<embedded-linux-dir>/examples/` entsteht der ausführbare Binärcode für den ARM-Prozessor:

```
$ cd <embedded-linux-dir>/examples/helloworld
$ make
gcc -O2 -c main.c -o main.o
gcc -O2 main.o -o helloworld
$ file helloworld
```

<sup>9</sup> Unter <http://support.ntp.org/bin/view/Servers/StratumTwoTimeServers> ist eine Liste von ntp-Zeitservern verfügbar.

```
helloworld: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux
      2.4.1, dynamically linked (uses shared libs), for GNU/Linux 2.4.1, not
      stripped
$ ./helloworld
Hello World!
```

Bei der Übersetzung größerer Projekte kann die Verwendung eines Cross Compilers auf dem Host-PC einige Zeit sparen. Für OpenWrt wurde ein solches Build-System bereits in Kapitel 3 vorgestellt. Leider steht aber für die NSLU2 kein vergleichbares Build-System zur Verfügung, und es wäre relativ viel Handarbeit notwendig, selbst ein solches zu erstellen. Eine einfachere Lösung ist, auf bestehende Werkzeuge anderer Systeme zurückzugreifen. So besitzt bspw. der OpenRISC Alekto in Kapitel 5 die gleiche Architektur, und entsprechend kann dessen Toolchain auch für die NSLU2 verwendet werden. Abschnitt 5.4 erklärt die Anwendung zur Erzeugung von ausführbarem ARM-Code mit einem x86-basierten PC.

## 4.5 NSLU2 als Druckerserver

Bei einem Betrieb der NSLU2 als Heimserver kann auch ein Druckerserver-Dienst eingerichtet werden. Hiermit kann ein bereits vorhandener USB-Drucker mehreren Rechnern zugänglich gemacht werden. CUPS ist ein plattformübergreifender Druckdienst für UNIX-Umgebungen und als Client-Server-Anwendungen für Linux und Mac OS X verfügbar [CUPS 08]. CUPS basiert zudem auf IPP<sup>10</sup>, das auch von Windows 2000 aufwärts unterstützt wird. Somit steht auch einem Mischbetrieb nichts im Wege.

Wenn nicht bereits bei der Systeminstallation im Auswahlmenü die Option **PrintServer** gewählt wurde, so wird der CUPS-Server unter Debian über folgende Pakete nachinstalliert:

```
$ sudo apt-get install cupsys cupsys-bsd
```

Um CUPS auch von außen erreichbar zu machen, muss der Port 631 freigeschaltet werden. Weiterhin muss *Browsing* aktiviert sein. Dazu sind in der CUPS-Konfigurationsdatei `/etc/cups/cups.conf` folgende Einträge vorzunehmen:

```
# Only listen for connections from the local machine.
# Listen localhost:631
Port 631
Listen /var/run/cups/cups.sock

# Show shared printers on the local network.
Browsing On
BrowseOrder allow,deny
BrowseAllow @LOCAL
```

<sup>10</sup> Abkürzung für *Internet Printing Protocol*, ein auf HTTP basierender Netzwerk-Druckdienst.

Die Zugriffsbeschränkungen auf den CUPS-Server müssen innerhalb des lokalen Heimnetzes üblicherweise weniger streng gehandhabt werden. Durch folgenden Eintrag wird allen Rechnern im lokalen Subnetz (repräsentiert durch @LOCAL) Zugang gewährt:

```
# Restrict access to the server...
<Location />
  Order allow,deny
  Allow localhost
  Allow @LOCAL
</Location>
```

Für den Administrator-Abschnitt `Location /admin` können die gleichen Einstellungen verwendet werden. Um auf eine SSL-Verschlüsselung und die damit verbundene zeitaufwändige Zertifikaterstellung zu verzichten, wird die folgende Zeile eingefügt:

```
DefaultEncryption Never
```

Falls nur eine bestimmte Benutzergruppe Rechte zur Änderung der Konfigurationsdatei über die Web-Oberfläche bekommen soll, so kann eine Zugriffsbeschränkung auf @SYSTEM gesetzt werden:

```
# Restrict access to configuration files...
<Location /admin/conf>
  AuthType Basic
  Require user @SYSTEM
  Order allow,deny
  Allow localhost
  Allow @LOCAL
</Location>
```

Die Systembenutzer müssen dazu der in der Variablen `SystemGroup` festgelegten Gruppe hinzugefügt werden:

```
# Administrator user group...
SystemGroup lpadmin
```

Dies geschieht an der Kommandozeile über folgenden Aufruf:

```
$ sudo adduser <benutzername> lpadmin
```

Die neue Konfiguration wird dann in `cupsys` übernommen:

```
$ /etc/init.d/cupsys reload
```

Danach kann der CUPS-Server über eine Web-Oberfläche erreicht und einfach verwaltet werden. In der Adresszeile wird nach der NSLU2-IP der in `/etc/cups/cups.conf` eingestellte CUPS-Port angegeben:

```
http://<nslu2-ip>:631
```

Über das Web-Frontend lassen sich nun neue Drucker hinzufügen und verwalten. Um einen neuen lokalen Drucker an die NSLU2 anzubinden, ist der Drucker zunächst über USB anzuschließen und einzuschalten. Dann wird auf der Konfigurationsseite *Drucker hinzufügen* ausgewählt und Name, Ort und Beschreibung eingegeben. Im folgenden Dialog sind die angeschlossenen Drucker in der Geräteliste aufgeführt, und es kann ein bestimmtes Modell ausgewählt werden. Die gängigen Treiber sind bereits in CUPS enthalten. Für exotischere Drucker wird eine PPD-Datei des Herstellers benötigt. Der Drucker sollte nun im CUPS-System aufgelistet werden (vgl. Abbildung 4.1).



Abb. 4.1. Web-Frontend zur Konfiguration des CUPS-Servers auf der NSLU2.

Um den Dienst nun von anderen Linux-Rechnern aus nutzen zu können, müssen dort die Pakete `cupsys-client` und `cupsys-bsd` installiert sein. Das Hinzufügen von Druckern geschieht unter Linux und Mac OS X über das Web-Frontend des Clients durch Aufruf der folgenden URL:

```
http://localhost:631
```

Auf der Konfigurationsseite wird die Aktion *Drucker hinzufügen* ausgewählt und wieder Name, Ort und Beschreibung eingegeben. Im folgenden Dialog ist als Protokoll IPP auszuwählen und dann die Druckeradresse anzugeben:

```
http://<nslu2-ip>:631/printers/<printer-name>
```

Mit der dann erfolgenden Treiberauswahl ist die Einrichtung des Netzwerkdruckers abgeschlossen. Unter Windows XP oder Vista kann ein CUPS-Drucker in der Druckerübersicht mit der Option *Drucker hinzufügen / Netzwerkdrucker hinzufügen* über die Angabe der URL eingebunden werden.



## 4.6 Weiterführende Hinweise

### 4.6.1 Erstellung eines vollständigen NSLU2-Backups

Für den Fall eines Datenverlustes ist es sinnvoll, neben einem Backup des USB-Sticks auch ein Backup des Flash-Speichers zu erstellen. Hierzu kann folgendermaßen vorgegangen werden: Zuerst wird in ein Verzeichnis auf dem USB-Stick oder auf der externen Festplatte gewechselt. Dann erfolgt die Kopie der sechs Partitionen des Flash-Speichers mithilfe des Tools `dd` in sechs Dateien:

```
$ dd if=/dev/mtdblock0 of=redboot.bin
$ dd if=/dev/mtdblock1 of=sysconf.bin
$ dd if=/dev/mtdblock2 of=kernel.bin
$ dd if=/dev/mtdblock3 of=ramdisk.bin
$ dd if=/dev/mtdblock4 of=flashdisk.bin
$ dd if=/dev/mtdblock5 of=fisdir.bin
```

Aus den einzelnen Dateien kann im Verlustfall leicht ein vollständiges Firmware-Image erstellt werden. Mithilfe des Befehls `cat` werden die Dateien zusammengeführt:

```
$ cat redboot.bin sysconf.bin kernel.bin ramdisk.bin flashdisk.bin fisdir.
bin > fullimage.bin
```

Die vollständige Binärdatei kann auf einfachem Wege über die Web-Oberfläche (zum Duplizieren einer NSLU2) oder das Programm `upslug2` (zur Wiederherstellung einer NSLU2) übertragen werden.

### 4.6.2 Einstellung der Taster-Funktion

Die Funktion des Tasters an der NSLU2 kann frei eingestellt werden. Die aktuelle Einstellung ist in der Datei `/etc/inittab` hinterlegt. Wenn z.B. gewünscht ist, dass das System bei Tastendruck keinen Neustart durchführt, sondern nur herunterfährt, so kann dies durch folgende Änderungen eingestellt werden:

```
# ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -h now
```

Durch einen Aufruf von `telinit q` werden die Einstellungen übernommen.

### 4.6.3 Probleme beim Booten

Sollte die NSLU2 auffallend lange zum Booten benötigen oder möglicherweise überhaupt nicht mehr hochfahren, so empfiehlt sich als erster Lösungsversuch

das Umstecken des Speichermediums und eine Überprüfung der Datei `/var/log/messages` mithilfe eines anderen Rechners.

Generell ist es ratsam, in der Datei `/etc/default/rcS` die Einstellungen für einen automatischen Start von `fsck`<sup>11</sup> zu setzen:

```
# FSCKFIX=no
FSCKFIX=yes
```

Bei einer evtl. notwendigen Überprüfung des Dateisystems wird damit die Notwendigkeit der Bestätigung durch Tastendruck „y“ vermieden, was ohne serielle Konsole ohnehin unmöglich wäre. Soll die Konsole zur Überwachung des Bootvorgangs oder zur Problembeseitigung verwendet werden, so kann diese durch folgende Maßnahmen auf den seriellen Port gelegt werden (vgl. Abschnitte 2.2 und 2.7):

In `/etc/inittab` werden dafür folgenden Einträge auskommentiert:

```
#1:2345:respawn:/sbin/getty 38400 tty1
#2:23:respawn:/sbin/getty 38400 tty2
#3:23:respawn:/sbin/getty 38400 tty3
#4:23:respawn:/sbin/getty 38400 tty4
#5:23:respawn:/sbin/getty 38400 tty5
#6:23:respawn:/sbin/getty 38400 tty6
```

Weiterhin sollten die Terminaleinstellungen in der gleichen Datei folgende Zeile enthalten:

```
T0:23:respawn:/sbin/getty -L ttyS0 115200 linux
```

Abschließend noch ein Hinweis: Lange Startzeiten von bis zu 20 Minuten können auf eine leere Pufferbatterie oder auf Kontaktschwierigkeiten in der Batteriehalterung hindeuten. Das Fehlen der Pufferung verhindert ein Laden des Treibermoduls für die I<sup>2</sup>C-Echtzeituhr beim Systemstart.

<sup>11</sup> *File System Check*. Programm zur Überprüfung und Reparatur von Unix-Dateisystemen.

## Debian auf dem Embedded-PC OpenRISC-Alekto

### 5.1 Einführung

Der OpenRISC-Rechner Alekto der Fa. VisionSystems wurde für den Einsatz im industriellen Umfeld konzipiert und wird mit einem vollständigen Debian-GNU/Linux-Betriebssystem für ARM geliefert [Vision Systems 08]. Damit ist automatisch auch eine umfangreiche, freie Software-Sammlung in Form von Debian-Paketen verfügbar und über die komfortable Debian-Paketverwaltung auch einfach zu nutzen. Die Verwendung von Debian als Basis für das Embedded-System hat zudem den Vorteil, dass der Anwender Erfahrungen aus PC-Distributionen wie Knoppix oder Ubuntu direkt nutzen kann, da diese auf dem gleichen Grundsystem beruhen. Die Verwendung von OpenWrt oder Puppy Linux erfordert hier etwas mehr Umdenken.

Üblicherweise wird der Alekto als industrieller Web-, Mail-, oder Druckerserver eingesetzt. Mit der Möglichkeit der Software-Entwicklung und der Vielzahl verfügbarer Schnittstellen lässt er sich aber auch als Schnittstellenwandler oder zur Umsetzung von Steuerungsaufgaben in der Automatisierungstechnik verwenden. Die Software-Entwicklung kann hierbei entweder auf dem Gerät selbst oder aber über die mitgelieferte Toolchain auf einem Host-Rechner erfolgen. Abschnitt 5.4 erklärt die beiden Möglichkeiten.

Aktuell wird für den Alekto ein Linux-Kernel mit Version 2.4.32 mitgeliefert. Dies bringt den Nachteil mit sich, dass insbesondere die Vorgehensweise bei der Treiberprogrammierung, wie sie in Kapitel 11 für Kernelversion 2.6 beschrieben ist, hier nicht direkt angewendet werden kann. Eine prototypische Portierung des Kernels 2.6 auf Alekto wurde nach aktuellem Stand aber bereits fertiggestellt, sodass dieses Problem zukünftig voraussichtlich nicht mehr existiert. Am Ende des Abschnitts 5.2 zur Systeminstallation wird auf die Verwendung verschiedener Kernelversionen eingegangen.

Die Hardware des Alekto wurde bereits in Abschnitt 2.5 vorgestellt. Das vorliegende Kapitel führt nun in die Installation des Basissystems, die Verwendung der Schnittstellen und die Software-Entwicklung für den Alekto ein. Zugehörige Beispiele sind im Verzeichnis `<embedded-linux-dir>/examples/alekto/` abgelegt.

## 5.2 Angepasste Debian-Installation

Im weiteren Text wird das Aufspielen eines ISO-Images des Betriebssystems beschrieben. Tatsächlich kann der Alekto optional mit einer CF-Karte mit vorinstalliertem Basissystem bzw. vollständigem System-Image erworben werden. In dieser Variante ist der Rechner sofort einsatzbereit und der vorliegende Abschnitt kann übersprungen werden. Für die Verwendung eigener CF-Karten oder auch für eine Neuinstallation wird in den nachfolgenden Schritten erklärt, wie fertige System-Images aufgespielt werden.<sup>1</sup> Abschnitt 5.8 stellt eine weitere Möglichkeit vor, eine Debian-Installation von Grund auf über den Debian-Installer aufzusetzen.

Die CF-Karte sollte zunächst über einen USB-Adapter an den Ubuntu-Hostrechner angeschlossen werden. Falls bereits eine Partition vorhanden ist und die Karte automatisch gemountet wird, muss diese zunächst wieder ausgehängt werden. Über das Programm *gparted*<sup>2</sup> werden auf der leeren Karte zunächst zwei Partitionen angelegt: Eine Systempartition vom Typ `ext2`<sup>3</sup> und dahinter liegend eine Partition für virtuellen Speicher vom Typ `swap`. Die Größe der Flashkarte sollte mindestens 2 GB betragen. Als Partitionsgrößen empfehlen sich für eine Karte dieser Größe 1,8 GB für das System und 256 MB für Swap-Speicher. Größere Karten werden mit ähnlichem Verhältnis partitioniert. Für die Systempartition ist abschließend noch das Boot-Flag zu setzen, dann kann *gparted* beendet werden. Der Datenträger sollte, falls nun vom System automatisch gemountet, wiederum ausgehängt werden.

Nun kann entweder das Basissystem oder das Komplettsystem auf die CF-Karte übertragen werden (aktuell handelt es sich um die Dateien `11022008_etch_base.bin` bzw. `27032008_etch_base.bin` auf der mitgelieferten DVD). Das Basissystem enthält bereits die Grundfunktionalität für

<sup>1</sup> System-Images werden zum aktuellen Zeitpunkt von VisionSystems nur für Kernel 2.4 geliefert.

<sup>2</sup> Gegebenenfalls über *Anwendungen/Hinzufügen* nachzuinstallieren.

<sup>3</sup> `ext2` ist etwas performanter als ein *Journaling*-Dateisystem wie `ext3`. Auf einem Standard-PC ergibt sich kaum ein Unterschied, auf einem Embedded-System mit magerer Prozessorleistung macht sich dies schon eher bemerkbar. Wird mehr Wert auf die Datensicherheit im Falle eines Absturzes gelegt, so sollte `ext3` verwendet werden. Es ist aber zu bedenken, dass sich die häufigeren Schreibzugriffe nachteilig auf die Lebensdauer des Flash-Speichers auswirken.

Software-Entwicklung (gcc-3.3, vim-tiny Editor) und Netzwerkdienste (SSH, TELNET, VSFTP, NETCAT). Das Komplettsystem ergänzt den Leistungsumfang um Dienste wie Samba, Apache2 Webserver, Courier Mailserver oder NTP. Über die Debian-Paketverwaltung können diese Anwendungen aber auch bei Bedarf rasch nachinstalliert werden, sodass eine Basisinstallation immer auch später noch zum Komplettsystem ausgebaut werden kann.

Um das Basissystem aufzuspielen, wird nun in das Verzeichnis mit den Image-Dateien gewechselt und im Terminal folgender Befehl eingegeben:<sup>4</sup>

```
$ dd if=11022008_etch_base.bin of=/dev/sdg
```

Wenn nicht klar ist, welche Gerätedatei (hier `/dev/sdg`) der CF-Karte zugeordnet ist, so sollte dies über *System/Einstellungen/Hardwareinformationen* nachgeprüft werden. Dort wird der Gerätenamen im Reiter *Erweitert* angezeigt. Vorsicht: Besonders, wenn andere externe Geräte wie USB-Speichersticks oder USB-Festplatten angeschlossen sind, ist hier Vorsicht angebracht. Wenn versehentlich auf ein solches Gerät zugegriffen wird, wird dieser Datenträger unbrauchbar.

Da das Dateisystem nach dem Aufspielen mit `dd` lediglich die Größe der Image-Datei besitzt, muss diese zunächst wieder über den Befehl `resize2fs`<sup>5</sup> korrigiert werden. Zuvor wird ein Check des Dateisystems verlangt, insgesamt sind dazu folgende Befehle notwendig:

```
$ sudo e2fsck -f /dev/sdg1
$ sudo resize2fs /dev/sdg1
```

Das Ergebnis lässt sich durch einen Aufruf von `df -h` überprüfen.

Die CF-Karte ist nun zum Booten vorbereitet und kann in den Slot des Alekto eingesetzt werden. Soll eine andere Kernelversion als die auf der System-DVD mitgelieferte verwendet werden, so muss lediglich die Datei `/boot/zImage` auf der CF-Karte ersetzt werden.

Unter `ftp://ftp.visionssysteme.de/pub/multiio/others/` stehen fertig übersetzte Kernel-Dateien zum Download bereit. Die Kernel-Quellen sind im SVN-Repository der Fa. VisionSystems verfügbar unter `http://svn.visionssysteme.de/`.

## 5.3 Erste Schritte

Ein großer Vorteil des Alekto ist der nach außen geführte Terminalstecker. Durch den Anschluss an einen Hostrechner lassen sich über die serielle Konsole

<sup>4</sup> Das Programm *Disk Dump* (`dd`) kopiert Daten 1:1 zwischen Quelle und Ziel und eignet sich zum Erstellen von Sicherungskopien kompletter Partitionen.

<sup>5</sup> Enthalten im Debian-Paket `ext2resize`.

das BIOS editieren, der Bootvorgang nachvollziehen und Netzwerkeinstellungen vornehmen. Es kann entsprechend auch ohne Einbindung in das Hausnetz ein komfortabler Zugriff auf den Embedded-PC erfolgen.

Auf dem Ubuntu-basierten Hostrechner können die Ausgaben auf die serielle Schnittstelle mit dem Programm *Serial Port Terminal* angezeigt werden. Dies ist über die Menüpunkte *Anwendungen/Hinzufügen* zu installieren und erscheint nach Auswahl *Zeige: Alle verfügbaren Anwendungen* in der Liste.

Die Verbindung zwischen Alekto und Hostrechner wird über das mitgelieferte Terminalkabel hergestellt. Dann wird das Programm *Serial Port Terminal* gestartet und dort im Menü *Configuration/Port* der serielle Port ausgewählt. Normalerweise ist dies `/dev/ttyS0` oder `/dev/ttyS1`. Bei Verwendung eines USB-Seriell-Adapters erscheint die Schnittstelle üblicherweise als `/dev/ttyUSB0`.<sup>6</sup> Abschließend sind folgende Übertragungsparameter einzustellen: 115 200 Baud, 8 Datenbits, 1 Stopbit, keine Parität, keine Flusssteuerung. Wird nun die Stromversorgung angeschlossen, so werden die Meldungen des Bootvorgangs auf der Konsole ausgegeben. Dann erscheint der folgende Anmeldebildschirm:

```
INIT: Entering runlevel: 2

Starting system log daemon: syslogd.
Starting kernel log daemon: klogd.
Starting portmap daemon...Already running..
Starting internet superserver: inetd.
Starting OpenBSD Secure Shell server: sshd.
Starting FTP server: vsftpd.
Starting NFS common utilities: statd.
Starting periodic command scheduler: crond.
Starting web server (apache2)....

Debian GNU/Linux 4.0 debian console

debian login:
```

Nun kann man sich als Benutzer `root` (Passwort: `linux`) anmelden, sollte danach aber über den Befehl `passwd` ein eigenes Passwort setzen. Für reguläre Benutzer steht außerdem ein Zugang mit Namen `user` zur Verfügung (Passwort `user`). Im nächsten Schritt empfiehlt es sich, die Netzwerkverbindung einzurichten, um zukünftig auf das Terminal verzichten zu können. Dies geschieht genau wie für ein Standard-Debian-System gemäß der Anleitung in Anhang A.3.1. Als Editoren sind von Haus aus `vim` oder `nano` verfügbar (vgl. Anhang A.2.2).

Wurde der Alekto mit dem Hausnetzwerk verbunden, so sollte nun auch ein SSH-Login vom Host-PC aus möglich sein:

```
$ ssh root@<alekto-ip>
```

<sup>6</sup> An dieser Stelle kann es notwendig sein, die Zugriffsrechte für die seriellen Geräte anzupassen. Näheres dazu ist in Anhang A.4.4 erklärt.

Um die häufigen Eingaben des Passwortes zu sparen, können für die regulären Benutzer SSH-Schlüssel-Paare erzeugt und der öffentliche Schlüssel mit dem Host-PC getauscht werden (vgl. Abschnitt A.3.2). Funktioniert die Internet-Verbindung von Alekto aus (Test über `ping google.de`), dann kann nun eine Aktualisierung der Paketinformationen erfolgen:

```
$ apt-get update
```

Jetzt lassen sich weitere, häufig benötigte Pakete installieren:

```
$ apt-get install sudo subversion bzip2 ncurses-dev
```

Sowohl der bereits eingerichtete Benutzer **user** als auch neue Benutzer können nun der Sudoers-Liste in `/etc/sudoers` hinzugefügt werden, um kurzzeitig für Administrationsaufgaben **root**-Rechte zu erwerben (vgl. Anhang A.4.5).

## 5.4 Software-Entwicklung

Es existieren zwei Möglichkeiten, Software für den Alekto zu entwickeln. Die erste Möglichkeit ist die native Entwicklung direkt auf dem Gerät selbst – der Ablauf ist dann genauso wie bei anderen Debian-Systemen. Möglich wird dies durch den umfangreichen Debian-Port und den ausreichenden Speicherplatz auf der CF-Karte. Zu bedenken ist aber, dass besonders der C++-Compiler auf dem 166 MHz-System nur sehr gemächlich arbeitet. Entsprechend ist diese Variante nur für kleinere Software-Projekte zu empfehlen. Der Ablauf ist wie folgt: Zuerst wird das Paket **build-essential** auf dem Alekto installiert:

```
$ apt-get install build-essential
```

Dieses Paket enthält eine Liste von Werkzeugen, welche zur Software-Entwicklung benötigt werden. Dazu zählen unter anderem die GNU-Version von **make**, die Entwicklungsbibliotheken der C-Bibliothek und natürlich die C- und C++-Compiler, sodass nun sofort die in `<embedded-linux-dir>/examples/` mitgelieferten Beispiele übersetzt werden können, z. B.:

```
$ cd <embedded-linux-dir>/examples/sockets/udp_cpp_receiver/
$ make
```

Wie bereits angesprochen, ist die native Entwicklung auf dem Gerät eher für kleinere Projekte geeignet. Daher wird nun eine zweite Möglichkeit der Software-Entwicklung vorgestellt.

Für die Übersetzung größerer Software-Projekte oder auch eines neuen Kernels (vgl. Abschnitt 5.7) ist es sinnvoll, eine Cross-Compiler-Toolchain auf einem schnellen Desktop-PC einzusetzen. Hierzu wird die von der Fa. VisionSystems

bereitgestellte Toolchain<sup>7</sup> in das Verzeichnis `/opt/` entpackt. Dann wird das Verzeichnis `/opt/arm-linux-gcc-4.2.2/bin/` zur Systemvariablen `PATH` hinzugefügt. Die Cross Compiler sind von nun an als `arm-linux-<compilername>` an der Kommandozeile verfügbar:

```
$ export PATH=/opt/arm-linux-gcc-4.2.2/bin/:$PATH
```

Um die mitgelieferten Beispiele auf dem PC-System mit einem Cross Compiler zu übersetzen, müssen diese für C und C++ im oberen Teil der Datei `<embedded-linux-dir>/src/Makefile.config` eingetragen werden:

```
CXX_STD      = g++
CXX_CROSS    = arm-linux-g++
...
CC_STD       = gcc
CC_CROSS     = arm-linux-gcc
```

Das eben genannte Beispiel kann dann über folgende Aufrufe auf einem PC für das ARM-Zielsystem übersetzt und mittels SCP dorthin kopiert werden:

```
$ cd <embedded-linux-dir>/examples/sockets/udp_cpp_receiver/
$ make CROSS=1
$ scp receiver root@<alekto-ip>:
```

Unter Umständen tritt nun eine Fehlermeldung in der folgenden Art auf:

```
$ ./receiver: /usr/lib/libstdc++.so.6: version 'GLIBCXX_3.4.9' not found (
required by ./receiver)
```

Dies bedeutet, dass die Toolchain mit einer Version der Standard-C++-Bibliothek erstellt wurde, die sich von jener auf dem Alekto unterscheidet. Dies ist ein häufiges Problem bei der wechselweisen Erstellung und Ausführung von Anwendungen auf mehreren Systemen. Es kann jedoch relativ einfach behoben werden, indem die in der Toolchain verwendete Bibliothek auf den Alekto kopiert, und dort der symbolische Link aktualisiert wird:

```
$ scp /opt/arm-linux-gcc-4.2.2/arm-vscom-linux-gnu/sys-root/usr/lib/
libstdc++.so.6.0.9 root@<alekto-ip>:/usr/lib
--Konsolen-Wechsel zu Alekto (als Benutzer root)--
$ ln -s /usr/lib/libstdc++.so.6.0.9 /usr/lib/libstdc++.so.6
```

Nun sollte sich die mit dem Cross Compiler erstellte Anwendung problemlos starten lassen.

<sup>7</sup> Verfügbar als `toolchain_gcc.4.2.2.libc.2.3.6.tar.bz2` auf der von Vision-Systems mitgelieferten DVD. Alternativ kann die Toolchain von der Herstellerseite <ftp://ftp.vision systems.de/pub/multiio/others/> geladen werden.



## 5.5 Zugriff auf die Alekto-Hardware

Grundsätzlich gibt es zwei Möglichkeiten, die Hardware-Schnittstellen des Alekto anzusprechen. Im Verzeichnis `/proc/vsopenrisc/` sind virtuelle Gerätedateien abgelegt, über welche Schnittstellen konfiguriert und Daten gelesen oder geschrieben werden können. Im ersten Abschnitt wird die einfache Variante in Form eines direkten Zugriffs auf diese Gerätedateien mittels Konsolenbefehlen vorgestellt, im zweiten Abschnitt die Verwendung innerhalb eigener Anwendungen mithilfe der `ioctl()`-Funktionen.

### 5.5.1 Anwendung der /proc-Erweiterungen in der Konsole

Über den `echo`-Befehl können Daten in die Gerätedateien geschrieben und so Hardware-Komponenten angesprochen werden. Um bspw. den eingebauten Summer an- oder abzuschalten, müssen in der Konsole lediglich folgende Befehlszeilen eingegeben werden:

```
$ echo 1 > /proc/vsopenrisc/buzzer
$ echo 0 > /proc/vsopenrisc/buzzer
```

Mittels `cat` wird der Inhalt der Gerätedateien auf der Konsole angezeigt. Um den Status der LEDs auszugeben oder diese an- bzw. auszuschalten, wird folgendermaßen vorgegangen:

```
$ cat /proc/vsopenrisc/leds
$ echo GREEN > /proc/vsopenrisc/leds
$ echo green > /proc/vsopenrisc/leds
```

Etwas anders verhält es sich mit den digitalen Ein- und Ausgängen (GPIO). Über `/proc/vsopenrisc/gpio_ctrl` muss zunächst festgelegt werden, ob die jeweiligen Pins als Ein- oder Ausgänge verwendet werden sollen, bevor über `/proc/vsopenrisc/gpio_data` die eigentlichen Daten gelesen und geschrieben werden können. Auch wenn es sich nur um acht GPIO-Leitungen und damit um ein zu schreibendes Daten-Byte handelt, müssen dennoch immer Daten vom Typ `long`, also vier Bytes, übermittelt werden. Vor der Übertragung der Daten wird zuerst eine Maske übertragen. Sie gibt an, welche der nachfolgenden Daten-Bits zu berücksichtigen sind. Jede Vorgabe erfolgt demnach in der Form:

```
$ echo <mask> <value> /proc/vsopenrisc/gpio_<register>
```

Dies ist besonders dann relevant, wenn verschiedene Anwendungen auf diese Schnittstelle zugreifen und die jeweilige Ausgabe auf bestimmte Bits beschränkt werden soll. Folgende Befehlszeilen setzen das Kontrollregister für alle acht GPIO-Pins und konfigurieren die unteren vier Bits als Ausgang, die oberen vier als Eingang. Anschließend wird eine Maske gesetzt, um die Änderung auf die ersten beiden und die letzten beiden Pins zu beschränken

(0x000000C3). Aufgrund der Konfiguration werden beim Übertragen des Wertes 0x000000FF nur die Pins 0 und 1 auf logisch 1 gesetzt:

```
$ echo 0x000000FF 0x0000000F > /proc/vsopenrisc/gpio_ctrl
$ cat /proc/vsopenrisc/gpio_ctrl
$ echo 0x000000C3 0x000000FF > /proc/vsopenrisc/gpio_data
$ cat /proc/vsopenrisc/gpio_data
```

Für die seriellen Schnittstellen und den I<sup>2</sup>C-Bus ist diese Art der Ansteuerung aufgrund der höheren Komplexität und des verwendeten Protokolls nicht empfehlenswert. Hier sollte die Ansteuerung ausschließlich über `ioctl()`-Befehle erfolgen (vgl. hierzu den nächsten Abschnitt).

### 5.5.2 Zugriff über `ioctl()`-Befehle

Der Zugriff über `ioctl()`-Befehle ist der Ansteuerung über das `/proc`-Dateisystem vorzuziehen, falls eine Nutzung nicht nur innerhalb von `cronjobs` oder Bash-Skripten, sondern auch aus eigenen C- oder C++-Anwendungen heraus erfolgen soll. Darüber hinaus ist diese Variante schneller als über die Konsole. Notwendige Makros und Strukturen sind in der Datei `vsopenrisc.h` definiert.

Diese Datei muss gemeinsam mit einigen weiteren Header-Dateien eingebunden werden, die für Standardausgabe, Fehlerbehandlung oder Dateizugriff notwendig sind.<sup>8</sup>

Um aus einer Anwendung heraus auf die Gerätedateien zugreifen zu können, wird das jeweilige Gerät über den Befehl `open()` zunächst geöffnet, dann werden mittels `ioctl()` die einzelnen Register angesprochen. Für den Zugriff auf die GPIO-Pins wird eine Struktur vom Typ `gpio_struct` benötigt, die Maske und Daten enthält. Damit sieht das Kommandozeilenbeispiel aus dem vorigen Abschnitt in C-Quellcode folgendermaßen aus<sup>9</sup>:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include "vsopenrisc.h"

int main() {
    int fd;
    fd = open("/dev/gpio", O_RDWR);
    if (fd < 0) {
        perror("could not open /dev/gpio");
    }
}
```

<sup>8</sup> Die Datei `vsopenrisc.h` ist, zusammen mit einigen von VisionSystems gelieferten Beispielen, unter `/home/user/` oder im Alekto-Beispielverzeichnis verfügbar.

<sup>9</sup> Verfügbar als Beispiel `gpio` unter `<embedded-linux-dir>/examples/alekto/`.

```

    exit(1);
}

struct gpio_struct cmd_val;
cmd_val.mask = 0xff;
cmd_val.value = 0x0f;
if(ioctl(fd, GPIO_CMD_SET_CTRL, &cmd_val) == -1) {
    perror("ioctl: GPIO_CMD_SET_CTRL failed");
    exit(1);
}

unsigned long cmd_ret_val;
if(ioctl(fd, GPIO_CMD_GET_CTRL, &cmd_ret_val) == -1) {
    perror("ioctl: GPIO_CMD_GET_CTRL failed");
    exit(1);
}
printf("Value: 0x%02X\n\n", cmd_ret_val);

struct gpio_struct val;
val.mask = 0xc3;
val.value = 0xff;
if(ioctl(fd, GPIO_CMD_SET, &val) == -1){
    perror("ioctl: GPIO_CMD_SET failed");
    exit(1);
}
unsigned long ret_val;
if(ioctl(fd, GPIO_CMD_GET, &ret_val) == -1) {
    perror("ioctl: GPIO_CMD_GET failed");
    exit(1);
}
printf("Value: 0x%02X\n\n", ret_val);
}

```

Die vollständige GPIO-API ist im Benutzerhandbuch bzw. im Kernel-Quelltext unter `drivers/char/gpio_vsopenrisc.c` dokumentiert.

Wie bereits die Linksys NSLU2, so verfügt auch der Alekto über einen nativen I<sup>2</sup>C-Bus. Diese Schnittstelle ist nach außen geführt und kann verwendet werden, um IO-Erweiterungen, A/D-Wandler oder Schrittmotortreiber anzusprechen (vgl. hierzu auch die ausführliche Beschreibung in Abschnitt 8.3.2). Der I<sup>2</sup>C-Bus ist im System als Gerät `/dev/i2c-0` eingebunden und wird intern dazu verwendet, um mit einer Realzeituhr vom Typ Dallas DS1337 zu kommunizieren.

Das Beispiel `<embedded-linux-dir>/examples/iic/iic_native` zeigt die grundlegende Verwendung des I<sup>2</sup>C-Busses hinsichtlich Öffnen, Setzen der I<sup>2</sup>C-Adresse mittels `ioctl()` und Schreib- bzw. Lesezugriffen.

Anzumerken ist, dass bei den Schreib- und Lesezugriffen nur die Daten angegeben werden – die I<sup>2</sup>C-Adresse wird bereits vom Treiber am Anfang jeder I<sup>2</sup>C-Nachricht eingefügt. Wird im Quelltext des genannten Beispiels die Adresse `0x68` der Alekto-Realzeituhr eingestellt, so folgt nach dem Übersetzen und Ausführen (mit den entsprechenden Dateirechten) die Antwort:

```

$ sudo ./iic_native
write: 0x0
read: 0x29

```

Diese Ausgabe besagt, dass ein Daten-Byte mit Wert 0 erfolgreich zu Adresse 0x68 abgeschickt und ein Daten-Byte mit Wert 0x29 empfangen wurde (der Rückgabewert kann variieren). Um eine sinnvolle Zeitanfrage zu starten und auszuwerten, müsste das Protokoll des Bausteins implementiert sein. An dieser Stelle interessiert aber zunächst nur die grundlegende Funktionalität. Weitere Informationen zum I<sup>2</sup>C-Bus und zum I<sup>2</sup>C-Protokollaufbau werden in Kapitel 8 gegeben. Kapitel 9 führt in eine I<sup>2</sup>C-Bibliothek zum Ansteuern einer Vielzahl von Bausteinen ein und erklärt die einzelnen Komponenten und deren Protokolle.

Weiterhin sei für die Nutzung der seriellen Schnittstellen für verschiedene Betriebsmodi wie RS-232, RS-422 oder RS-485 auf das ausführliche Beispiel in Kapitel 6.5.4 des Alekto-Benutzerhandbuchs verwiesen. Eine Verwendung der RS-232-Schnittstelle mit komfortablem C++-Interface wird im vorliegenden Buch in Abschnitt 7.2 beschrieben.

## 5.6 Watchdog-Timer

Der Alekto verfügt über einen Timer-Baustein, welcher nach Ablauf einer einstellbaren Zeit einen Hard-Reset durchführt und sich so ab Kernel-Version 2.6 als Watchdog-Timer<sup>10</sup> nutzen lässt (vgl. Abschnitt 1.3.7). Da beim Alekto der Bootvorgang relativ lange dauert, eignet sich der Watchdog-Timer nicht unbedingt zur Absicherung sicherheitskritischer Anwendungen. Seine Aktivierung kann aber zumindest sicherstellen, dass das Gerät nach einem Systemabsturz wieder in den Ursprungszustand versetzt wird und erreichbar bleibt.

Um den Watchdog nutzen zu können, muss zunächst ein neues Gerät erstellt werden. Über dieses wird dann der Timer-Baustein mittels `ioctl()`-Befehlen konfiguriert. Folgender Aufruf erzeugt ein zeichenorientiertes Gerät mit Hauptgerätenummer 10 und Untergerätenummer 130:

```
$ mknod /dev/watchdog c 10 130
```

Durch Angabe der Gerätenummern kann eine Verbindung zwischen dem Gerät und den entsprechenden Gerätetreibern im Kernel hergestellt werden. Eine Auflistung aller Gerätenummern findet sich in den Kernel-Quellen<sup>11</sup> im Verzeichnis `Documentation/devices.txt`. Weitere Informationen zu `mknod` werden in Anhang A.4.3 gegeben. Das Beispiel `watchdog` zeigt die generische Verwendung der Watchdog-API unter Linux. Sofern die notwendige Hardware vorhanden ist, kann dieses Beispiel auch für andere Embedded-Plattformen verwendet werden.

<sup>10</sup> Ein *Watchdog* ist eine Komponente zur Überwachung der korrekten Funktion anderer Systemteile.

<sup>11</sup> Die gepatchten Alekto-Kernel-Quellen sind im Subversion-Verzeichnis von Vision-Systems unter <http://svn.visionssysteme.de/> verfügbar.

Die Vorgehensweise ist wie folgt: Zuerst wird die Header-Datei `#include <linux/watchdog.h>` eingebunden, dann wird der Watchdog-Timer geöffnet:

```
// open watchdog device
int fd = open("/dev/watchdog", O_RDWR);
```

Folgender Befehl gibt einen Zeitwert in Sekunden vor und startet den Timer:

```
// set watchdog timeout
int timeout = 5;
ioctl(fd, WDIOCSSETTIMEOUT, &timeout);
```

Um den Timer auf den bereits eingestellten Wert zurückzusetzen, wird `ioctl()` mit dem Argument `WDIOCKEETIMOUT` aufgerufen:

```
ioctl(fd, WDIOCKEETIMOUT, NULL);
```

Falls der Watchdog beim Beenden der überwachenden Anwendung wieder abgeschaltet werden soll, so geschieht dies über einen `ioctl()`-Aufruf mit dem Argument `WDIOCSSETOPTIONS` und entsprechend gesetzten Flags:

```
int flags = 0;
flags |= WDIOCS_DISABLECARD;
ioctl(fd, WDIOCSSETOPTIONS, &flags);
```

Weitere Details zur Watchdog-API können in den Quelltext-Dateien des Kernels für Watchdogs vom Typ Kendin/Micrel KS8695 nachgelesen werden.<sup>12</sup> Im mitgelieferten Beispiel können als Argumente Zeitwerte für Wartezeit (`sleep_time`) und Überwachungszeit (`timeout`) mitgegeben werden. In einer Schleife wird zyklisch pausiert bzw. ein `keep_alive` ausgeführt, um den Alekto am Laufen zu halten. Liegt die Wartezeit über der Überwachungszeit oder wird über die Direktive `#define KEEPALIVE` der Rücksetz-Vorgang abgeschaltet, dann wird ein Hard-Reset ausgeführt.

## 5.7 Erstellung eines eigenen Alekto-Kernels

Werden spezielle Anforderungen an einen Linux-Kernel gestellt, so kann die Erstellung eines eigenen Kernels notwendig werden. Die umfangreichen Kernel-Quellen sollten aus Zeitgründen nicht auf dem Alekto selbst übersetzt werden, sondern auf einem schnelleren PC-System. Dies geschieht mithilfe der von VisionSystems erstellten Toolchain. Die Installation erfolgt wie in Abschnitt 5.4 beschrieben.

Im zweiten Schritt werden die Kernel-Quellen heruntergeladen. Da es sich um eine auf die Alekto-Hardware angepasste bzw. gepatchte Version handelt, wird

<sup>12</sup> Zu finden in den Kernel-Quellen unter `drivers/watchdog/ks8695_wdt.c`.

diese nicht von **kernel.org** bezogen, sondern aus dem SVN-Repository des Herstellers:<sup>13</sup>

```
$ svn checkout svn://svn.visionssysteme.de/linux-2.6-stable/trunk <kernel\_src\_dir>
```

Nun wird in **<kernel\_src\_dir>** der Kernel-Konfigurator gestartet:<sup>14</sup>

```
$ make menuconfig
```

Nach der Konfiguration wird das Menü beendet und die Einstellungen werden abgespeichert. Jetzt kann der Kernel übersetzt werden:

```
$ make dep
$ make zImage
```

Nach einem etwas längeren Übersetzungsdurchlauf liegt der fertige Kernel in Form einer Datei **zImage** im Verzeichnis **arch/arm/boot**. Die Kernel-Datei kann nun auf der CF-Karte unter **/boot/** abgelegt werden, muss dort nach einer Sicherungskopie des aktuellen Kernels jedoch in **vmlinux** umbenannt werden. Nach einem Neustart des Systems wird der neue Kernel dann automatisch verwendet.

## 5.8 Vollständige Debian-Installation

Mitunter kann es sinnvoll sein, auf dem Alekto ein Debian-Linux von Grund auf neu zu installieren. Dies gilt insbesondere dann, wenn der Anwender ein besonders schlankes System ohne die im Basis-Image enthaltenen Pakete erstellen möchte (bspw. auch für Testzwecke).

Für eine Installation werden grundlegende Funktionen wie die Unterstützung für DVD-Laufwerke, Netzwerk oder das Lesen und Schreiben von Dateisystemen benötigt. Entsprechend muss zunächst ein Kernel auf der CF-Karte vorhanden sein. Weiterhin wird ein *Installer* benötigt, der durch die Installation führt. Zusammengenommen werden zur Vorbereitung der Installation einer **ext2**- oder **ext3**-Partition folgende Dateien auf die CF-Karte kopiert:

- **init.rd** – RAM Disk Image mit Debian Installer,
- **zImage** – Der Linux-Kernel,
- **kparam** – Datei mit Kernelparametern: **mem=0x3b00000 root=/dev/ram** (nutzbarer Speicher: 59 MB, Kernel auf RAM Disk).

<sup>13</sup> Dieses Verzeichnis ist derzeit aktuell; das Repository des Herstellers kann über <http://svn.visionssysteme.de/> nach neueren Versionen durchsucht werden.

<sup>14</sup> Hierfür muss die Bibliothek **ncurses** installiert sein, was über **sudo apt-get install libncurses5-dev** nachgeholt werden kann.

Diese Dateien sind speziell auf die Alekto-Hardware zugeschnitten und in den Kernel-Quellen `<kernel_src_dir>/Vscom/installation` enthalten (vgl. voriger Abschnitt). Andere im Debian-Port vorhandene Debian-Installer für Arm-Prozessoren der Typen `iop32x`, `ixp4xx` oder `netwinder` funktionieren leider nicht. Darüber hinaus besteht auch die Möglichkeit, Debian über ein freigegebenes Windows-Verzeichnis zu installieren. Hierzu sei auf Abschnitt 9.1 des Handbuchs verwiesen.<sup>15</sup>

Die Installation erfolgt über den seriellen Konsolenanschluss des Alekto. Dafür ist das mitgelieferte Kabel mit dem Hostrechner zu verbinden und das *Serial Port Terminal* zu starten (vgl. Abschnitt 5.3). Weiterhin werden ein externes USB-CD-Laufwerk und die mitgelieferte Debian 4.0-Installations-CD benötigt. Wurden die drei Dateien mit einem PC auf die CF-Karte kopiert, diese in den Alekto eingesteckt, ein Laufwerk samt eingelegter Debian-CD angeschlossen und die Konsole mit dem PC verbunden, so kann der Alekto angeschaltet werden.

Der Installer wird sich zunächst mit der Ausgabe melden, dass der *Low-Memory-Mode* benutzt wird; bei nur 32 MB RAM eine sinnvolle Entscheidung. Sollte im nächsten Schritt kein Laufwerk erkannt werden, so ist die Frage nach dem Laden der Treiber von Diskette zu verneinen. In diesem Fall muss das CD-Laufwerk manuell als `/dev/cdrom` ausgewählt werden.

Nun folgt die Meldung, dass keine Kernelmodule gefunden wurden. Dies ist korrekt und hängt damit zusammen, dass alle Module in den Kernel kompiliert sind. Die Frage nach dem Fortsetzen kann entsprechend mit *Ja* beantwortet werden. Jetzt wird einer der beiden Netzwerkanschlüsse ausgewählt und manuell oder per DHCP konfiguriert – die Netzwerkverbindung wird zu einem späteren Zeitpunkt benötigt. Eine geführte Partitionierung ist für das doch etwas spezielle System nicht zu empfehlen, stattdessen sollte im Partitionierungsschritt die manuelle Methode ausgewählt werden. Die Größen der Partitionen sind ähnlich wie in Abschnitt 5.2 anzugeben: Bei einer CF-Karte mit 2 GB Gesamtspeicher entfallen ca. 1,8 GB auf das System (`ext2` oder `ext3`) und 256 MB auf den virtuellen Swap-Speicher (vgl. Abschnitt 5.2). Abschließend wird das Boot-Flag für die Systempartition gesetzt und der *Mount Point* wird auf das Wurzelverzeichnis `/` gelegt. Nun werden beide Partitionen formatiert, im nächsten Dialogfenster ein Passwort für `root` festgesetzt und ein weiterer Benutzer angelegt.

Da keine offizielle Unterstützung für Alekto im Debian-Port existiert, kann zunächst kein Kernel mitinstalliert werden. Die Frage, ob ohne Kernel fortgefahren werden soll, ist demnach mit *Ja* zu beantworten. Nun wird ein Debian-Mirror ausgewählt, von welchem zukünftig Informationen und Pakete bezogen werden können und nach Belieben an der Erfassung der Paketnutzung teilgenommen. Bei der Software-Auswahl reicht das Standard-System vollkommen

<sup>15</sup> Zu finden unter `<embedded-linux-dir>/datasheets/OpenRISC_User_Manual.pdf`.

aus. Drucker-, Mail- oder Webserver können auch problemlos nachträglich installiert werden. Auf keinen Fall sollte das sehr umfangreiche *Desktop Environment* ausgewählt werden, da das X-Window-System den Alekto völlig überfordern würde.

Die Installation ist nun abgeschlossen. Ein Boot-Loader ist für Alekto nicht verfügbar, die nächste Meldung ist deshalb einfach mit *weiter* zu bestätigen. Nach dem darauffolgenden Neustart ist der Alekto sofort abzuschalten. Da auf dem System bisher noch kein Kernel installiert wurde, wären weitere Boot-Versuche erfolglos (der ursprüngliche Kernel auf der CF-Karte wurde mittlerweile überschrieben). Die CF-Karte wird nun entnommen und in den PC eingesteckt. Dann werden folgende Befehle ausgeführt, um den Kernel zu kopieren und die Kernelparameter abzulegen:<sup>16</sup>

```
$ sudo cp <kernel_src_dir>/arch/arm/boot/zImage /media/disk/
$ echo mem=59M root=/dev/hda1 > kparam
$ sudo cp kparam > /media/disk/boot/
```

Die Konsolenschnittstelle ist in diesem ursprünglichen Zustand noch nicht aktiviert. Dies geschieht, indem in der Datei */etc/inittab* die Schnittstelleneinträge für *tty1* bis *tty6* auskommentiert werden und die Konsole auf eine Baudrate von 115 200 bit/s eingestellt wird:<sup>17</sup>

```
# Note that on most Debian systems tty7 is used by the X Window System,
# so if you want to add more gettys go ahead but skip tty7 if you run X.
#
#1:2345:respawn:/sbin/getty 38400 tty1
#2:23:respawn:/sbin/getty 38400 tty2
#3:23:respawn:/sbin/getty 38400 tty3
#4:23:respawn:/sbin/getty 38400 tty4
#5:23:respawn:/sbin/getty 38400 tty5
#6:23:respawn:/sbin/getty 38400 tty6
# Example how to put a getty on a serial line (for a terminal)
#
T0:2345:respawn:/sbin/getty -L console 115200 linux
#T1:23:respawn:/sbin/getty -L ttyS1 9600 vt100
```

Nun kann das Laufwerk abgeschaltet und die CF-Karte in den Alekto eingesteckt werden. Beim nächsten Einschalten startet das jungfräuliche Debian. Nach der Anmeldung als *root* ist es sinnvoll, zuerst die Paketinformationen zu aktualisieren. Da von nun an alle Pakete über das Netzwerk und nicht mehr von der DVD installiert werden sollen, empfiehlt es sich, die entsprechenden Einträge mit *deb cdrom* aus der Quellenliste in */etc/apt/sources-list* zu entfernen. Dann kann die Aktualisierung der Paketinformationen und die Installation weiterer, oft benötigter Pakete erfolgen:

```
$ apt-get update
$ apt-get install sudo ssh subversion
```

Der Alekto ist nun einsatzbereit!

<sup>16</sup> Hierbei wird angenommen, dass die CF-Karte automatisch unter */media/disk* gemountet wurde.

<sup>17</sup> Diese Aktion muss als Benutzer *root* ausgeführt werden.



## Puppy Linux auf dem Embedded-PC MicroClient Jr./Sr.

### 6.1 Einführung

Puppy Linux ist eine Linux-Distribution, die besonders für ältere bzw. mobile PCs mit eingeschränkten Ressourcen ausgelegt ist.<sup>1</sup> Durch die Genügsamkeit hinsichtlich RAM- und Taktfrequenzbedarf ist Puppy aber auch gerade für Embedded-PCs gut geeignet. Eine Haupteigenschaft von Puppy ist, dass das gestartete System fast vollständig von einer RAM-Disk läuft. Mit dieser Eigenschaft spielt Puppy entsprechend auch besonders gut mit CF-Cards oder anderen Flash-Festspeichermedien zusammen. Der thailändische Distributor der MicroClient-PCs, die Fa. NorhTec, liefert die Geräte mittlerweile optional mit vorinstalliertem Puppy auf CF-Card aus. Wer hier etwas Geld sparen möchte, der kann diese Installation aber auch leicht selbst vornehmen.

Im nachfolgenden Text wird die Puppy-Installation von Puppy 3.01 erklärt, die Paketverwaltung wird vorgestellt und die Software-Entwicklung unter Puppy wird erläutert. Die Vorgehensweise ist hierbei für MicroClient Jr. und Sr. identisch – der Sr. ist allerdings wesentlich schneller (vgl. Abschnitt 2.4).

### 6.2 Puppy-Installation

Am Anfang der Puppy-Installation steht ein Download des aktuellen ISO-Images von der Puppy-Homepage. Hier findet sich auch eine gute Einführung in die Bedienung und die Besonderheiten von Puppy. Das ISO-Image wird nach dem Download auf eine CD gebrannt (Ubuntu: Rechtsklick auf die ISO-Datei ...). Die dieserart erstellte CD ist bootfähig und stellt ganz ähnlich wie Knoppix eine Live-Distribution dar; der Entwicklungs-PC kann entsprechend von dieser CD gebootet werden [Puppy Linux 08].

---

<sup>1</sup> <http://www.puppylinux.org>.

Um das Betriebssystem auch auf dem MicroClient nutzbar zu machen, muss es nun bootfähig auf die CF-Card kopiert werden. Hierfür existieren drei Möglichkeiten, die im Folgenden näher erläutert werden:

*a.) Puppy-Installation auf CF-Card mittels CF-IDE-Adapter*

Die CF-Card kann mittels eines CF-IDE-Adapters an den Entwicklungs-PC angeschlossen werden. Sie erscheint dann als Festplatte und wird als solche formatiert mit dem Tool *GParted: Menu / System / GParted Partition Manager*.

Im Menü von GParted ist auszuwählen: *Devices ...*, im Regelfall heißt die CF-Card */dev/sda* oder */dev/sdb*. Anzulegen ist eine große ext3-Partition, welche ca. 80 % des verfügbaren Speicherplatzes einnimmt und eine kleinere Swap-Partition, die den restlichen Platz nutzt. Dann wird das Boot-Flag auf der ersten Partition gesetzt und abschließend im Menü ausgewählt *Edit / Apply all Operations*. Danach kann nun mit dem Puppy Universal Installer eine Installation auf die neue CF-Card-Festplatte vorgenommen werden (*Menu / Setup / Puppy Universal Installer*). Wenn die Frage zum MBR kommt, so ist hier *default* zu wählen (zur Bezugsquelle für den CF-IDE-Adapter: vgl. Anhang E).

Eine Anmerkung: Bei Formatierungsaktionen sollte sich der Anwender lieber zweimal vergewissern, dass er auch das richtige Medium ausgewählt hat und nicht etwa versehentlich die Festplatte des Entwicklungs-PCs formatiert. Mit einem Blick auf die angegebene Größe des Mediums erhält man im Regelfall Gewissheit.

*b.) Installation mittels USB-CF-Adapter*

Die Installation auf einer CF-Card funktioniert auch ohne den angesprochenen Adapter. Ein einfaches USB-CF-Card-Lesegerät reicht aus, da der Puppy Universal Installer auch den Modus *USB CF Flash drive, later move CF to IDE* kennt. Die vorhergehende Formatierung läuft ab wie unter *a.)* beschrieben.

*c.) Installation mittels USB-Stick*

Die Installation kann auch auf dem MicroClient direkt vorgenommen werden. Da dieser aber kein Laufwerk besitzt, muss er von einem USB-Stick gebootet werden (nicht vergessen: im BIOS ist hierfür die Boot-Reihenfolge umzustellen). Zur Erstellung des bootfähigen Live-Puppy-USB-Sticks kann auf dem Entwicklungs-PC das freie Tool *unetbootin* verwendet werden. Es ist

sowohl für Linux, als auch für Windows erhältlich; die Bedienung ist selbsterklärend: <http://unetbootin.sourceforge.net>. Die vorher erforderliche Formatierung läuft ab wie unter *a.*).

Die erste und die dritte Variante funktionieren naturgemäß auch mit anderen Distributionen, wie z. B. XUbuntu. Falls bei dem Prozedere USB-Stick oder CF-Card Probleme bereiten, beispielsweise von *GParted* nicht mehr beschrieben werden können, so sollte man zuerst schauen, ob die Medien vielleicht gemountet sind. Wenn das nicht die Ursache ist, so hilft im Regelfall ein Aus- und erneutes Einstecken.

Nun kann der MicroClient von der CF-Card gebootet werden. Falls noch ein USB-Stick eingesteckt ist, so muss hierzu die Boot-Reihenfolge wieder entsprechend angepasst werden. Für die Grafikdarstellung hat sich bewährt: XORG, 1280×1024×24.

Mit der Verwendung von Auto DHCP ist die Netzwerkeinrichtung besonders einfach, allerdings muss hierfür anfänglich die MAC-Adresse des MicroClient-PCs ermittelt und in die DHCP-Tabelle des Heim-Routers eingetragen werden. Mit dem folgenden Aufruf an der Konsole wird die MAC-Adresse des MicroClients ausgegeben:

```
$ dmesg | grep RealTek
eth0: Realtek RTL8139 at 0xc80b0f00, 44:4d:50:b1:xx:xx, IRQ 5
```

Dann kann das Connect-Symbol auf dem Desktop geklickt, *connect by network interface* ausgewählt und Auto DHCP eingestellt werden. Abschließend sollte die Konfiguration gespeichert werden.

Puppy läuft komplett von einer RAM Disk. Zur Speicherung von Einstellungen und neuen Dateien wird beim Herunterfahren eine Datei **pup.save.2fs** angelegt. Entsprechend muss vor der ersten Verwendung neu gebootet werden: Erst dann existieren die 2fs-Datei und das Verzeichnis **/mnt/home**.

Anmerkung: Die vorgestellten Abläufe zur Installation eines Linux-Derivates auf CF-Card können nicht nur unter Puppy angewandt werden, sondern auch unter anderen Derivaten. Andere interessante und schlanke Linux-Derivate, die sich besonders für Embedded Devices eignen sind XUbuntu, Muppy, CoolCNC und MCPup. MCPup ist hierbei besonders interessant, da es speziell für den MicroClient angepasst ist und auch die Grafik-Hardwarebeschleunigung unterstützt.

## 6.3 Paket-Management unter Puppy

Puppy verwendet mittlerweile einen speziellen Paketmanager namens PetGet. Der Aufruf erfolgt mittels *Menü / Setup / Puppy Package Manager* oder an der Kommandozeile durch die Eingabe von **petget**. Das grafische Tool ist

selbsterklärend und ermöglicht die Installation von Paketen aus dem Netz oder auch von auf der Platte abgelegten Paketen. Weiterhin kann damit ein Check der Abhängigkeiten und auch die Deinstallation erfolgen. Der Info-Button im Start-Dialog liefert weiterführende Hilfeinformationen.

Neben PetGet wird auch das ältere DotPup-Format noch immer unterstützt. Nach dem Download aus dem Netz werden diese Pakete einfach im Puppy-Dateimanager ROX angeklickt. Dadurch erfolgt das Prüfen der Checksumme, das Entpacken und nach einer Rückfrage die Installation.

Folgende URLs sind gute Anlaufstellen auf der Suche nach bestimmten Software-Paketen für Puppy:

- <ftp://ibiblio.org/pub/linux/distributions/puppylinux>
- <http://www.puppylinux.org/home/links>
- <http://dotpups.de/dotpups>
- <http://www.murga-linux.com/puppy>
- <http://puppylinux.org/wiki/archives/old-wikka-wikki/everything-else/additionalprograms>

Eine weitere Möglichkeit, Puppy um eine bestimmte Software zu ergänzen, ist die Verwendung des sog. *pb\_debianinstallers*. Dieser muss über den PetGet-Paketmanager nachinstalliert werden und ermöglicht dann auch die Verwendung von Debian-Paketen. Vgl. hierzu auch <http://www.puppylinux.org>, Suchbegriff: Debian Programs.

Die vierte und letzte Möglichkeit, Puppy bequem um neue Software zu ergänzen, ist die Verwendung der sog. Squash File System-Dateien (.sfs-Dateien). Das Squash-Dateisystem stellt ein komprimiertes und schreibgeschütztes Dateisystem für Linux dar. Eine typische Anwendung ist ähnlich wie bei dem tar.gz-Format die Erstellung von Archiven.

In Puppy wird das sfs-Dateiformat für größere Programmpakete wie Compiler-Tools oder Kernel-Quellen genutzt. Die Anwendung ist wie folgt: Das gesuchte sfs-Paket wird aus dem Netz geladen, die md5-Checksumme wird kontrolliert und die Datei wird unter `/mnt/home` abgelegt (am gleichen Ort, an dem auch die gepackten Puppy-Systemdateien `pup_301.sfs` ... liegen).<sup>2</sup> Nach einem Bootvorgang wird das sfs-Paket automatisch installiert und Puppy fragt nach, ob dieses Paket nun zum Standard-Bootumfang gehören soll. Ein Aufruf des Befehls `fixmenus` an der Konsole ergänzt evtl. bereitgestellte Menüeinträge.

Ein Beispiel für diese Vorgehensweise folgt mit der Installation der Compiler-Tools im nächsten Abschnitt.

---

<sup>2</sup> Merke: `/mnt/home` wird durch den ersten Bootvorgang nach der Puppy-Installation angelegt.

## 6.4 Software-Entwicklung unter Puppy

Die Software-Entwicklung unter Puppy kann direkt auf dem MicroClient-Rechner erfolgen, da an diesem auch Tastatur und Bildschirm angeschlossen werden können. Oftmals wird der Rechner aber einfacher über SSH ferngesteuert. Im weiteren Text werden die SSH-Installation und die Installation der Entwicklungs-Tools vorgestellt.

### *a.) Open SSH-Installation*

OpenSSH gehört nicht zum Standard-Lieferumfang von Puppy, das Tool kann aber leicht nachinstalliert werden. Zuerst wird auf dem MicroClient die entsprechende .pup-Datei heruntergeladen von <http://dotpups.de/dotpups/Network>; im vorliegenden Fall handelt es sich um das Paket `Openssh-4.4p1.pup`. Die Installation erfolgt dann über den Dateimanager ROX wie bereits unter 6.3 beschrieben.

Nun können die SSH-Schlüssel generiert und der Daemon gestartet werden. Hierzu ist folgendes Skript aufzurufen:

```
$ /usr/local/openssh-4.4p1/sshd.sh
```

Damit der SSH-Daemon ab jetzt automatisch gestartet wird, muss noch eine Skript-Textdatei im Verzeichnis `/etc/init.d/` angelegt werden.<sup>3</sup> Der Name kann beliebig gewählt werden, der Inhalt besteht aus der Zeile `/usr/sbin/sshd`. Weiterhin ist das `,x'-Flag für executable zu setzen (chmod a+x sshd). Abschließend wird an den Root-User noch mittels passwd ein Passwort vergeben. ifconfig liefert die aktuell zugeteilte IP, mit welcher nun ein erster Einlog-Versuch vom Desktop-PC aus erfolgen kann:4`

```
$ ssh root@192.168.1.34
```

### *b.) Installation der Entwicklungs-Tools*

Die Software-Entwicklungs-Tools wie Compiler, Make und Kernel-Header-Files sind für Puppy zusammengestellt in `devx_301.sfs` und unter folgender URL zu beziehen:<sup>5</sup> <http://www.puppylinux.org/wiki/archives/old-wikka-wikki/categorydocumentation/compiling>.

<sup>3</sup> Standardmäßig sind unter Puppy die Editoren `vi` und `nano` nicht vorhanden. Man kann stattdessen aber `geany` verwenden.

<sup>4</sup> Zu Details bzgl. SSH und zum Umgang mit den Schlüsseln vgl. auch Abschnitt A.3.2.

<sup>5</sup> Die Versionsnummer muss zur Puppy-Version passen. Im vorliegenden Fall ist dies V 3.01.

Die anschließende Installation erfolgt wie im Abschnitt 6.3 erklärt. Nachdem der Anwender mit einem Editor wie z. B. **geany** ein kleines Hallo-Welt-Beispiel (vgl. `<embedded-linux-dir>/examples/helloworld/`) in C erstellt hat, kann ein erster Test erfolgen:

```
$ g++ hallo.c
$ ./a.out
Hallo Welt!
```

### *c.) Installation der Kernel-Quellen*

Eine Vorbemerkung: Wie bereits in Abschnitt 2.4 vorgestellt, existieren mehrere MicroClient-Derivate. Die preiswertesten Ausführungen besitzen nur 256 MB RAM, zu wenig, um einen Kernel zu bauen. Der hier vorgestellte Ablauf wurde mit einem MicroClient Sr. mit 1024 MB RAM durchgeführt (die Standardausstattung ist bei diesem PC 512 MB).

Sowohl um einen neuen Kernel auf dem MicroClient zu bauen, als auch, um neue Module wie bspw. das IO-Warrior-Modul nutzen zu können, müssen die Kernel-Quellen installiert werden. Die wichtigste URL hierfür ist <http://www.puppylinux.com/development/compilekernel.htm>.

Die dort beschriebene Einrichtung der Entwicklungsumgebung wurde bereits in Abschnitt 6.4 vorgestellt. Es folgt entsprechend direkt die Beschreibung der Installation der Kernelquellen (*Full Kernel Sources*). Die Kernelquellen liegen wieder als .sfs-Archiv vor und wieder ist auf die Versionsnummer zu achten. Im vorliegenden Fall – für Puppy 3.01 – ist die Datei **kernel-src.301.sfs** erforderlich. Der FTP-Server zum Download ist über die genannte Development-Seite zu erreichen (Link dort: *Download Page*). Genau wie bereits zuvor für die anderen .sfs-Archive beschrieben, so muss auch diese Datei unter `/mnt/home` abgelegt werden.<sup>6</sup> Anschließend ist ein Neustart erforderlich, weiterhin muss beim erneuten Starten das neue sfs-Paket in der erscheinenden Dialogbox hinzugefügt werden.

Nun muss die .config-Datei in das Verzeichnis `/usr/src/linux-<version>` kopiert werden. Falls hier bereits eine Datei vorhanden ist, so ist diese zuerst umzubenennen. Die .config-Datei sollte laut Puppy-Doku unter `/lib/modules` zu finden sein, in unserem Falle musste sie allerdings von folgender URL kopiert werden (Datei `D0Tconfig-<version>`, umzubenennen nach `.config`):

<http://puptrix.org/sources/kernel-2.6.21.7-pup300.7>

Folgende Schritte sind entsprechend erforderlich (je nach Version kann der genaue Verzeichnisnamen hiervon abweichen):

<sup>6</sup> Hier könnte es je nach CF-Kartengröße auch auf dem Festspeichermedium langsam eng werden – das ist entsprechend zu prüfen. Die sfs-Datei ist rund 60 MB groß.

<sup>7</sup> Die .config-Datei funktioniert für V3.00 und V3.01.

```
$ cd /usr/src/linux-2.6.21.7
$ mv .config .config.bak
$ wget http://puptrix.org/sources/kernel-2.6.21.7-pup300/DOTconfig-K2
  .6.21.7-8SEPT07
$ mv DOTconfig-K2.6.21.7-8SEPT07 .config
$ make menuconfig
$ make bzImage
```

Eine Anmerkung: Am einfachsten ist es, **menuconfig** nach den notwendigen Veränderungen per Exit zu verlassen, es erfolgt dann automatisch eine Abfrage zum Speichern, die zu bestätigen ist (der Menüpunkt zum Speichern funktioniert nicht).

Nach dem Übersetzen befindet sich der neu compilierte Kernel nun im Verzeichnis `/usr/src/linux-2.6.21.7/arch/i386/boot/bzImage`.

Die Erstellung der Module geschieht folgendermaßen:

```
$ cd /lib/modules
$ mv 2.6.21.7 2.6.21.7-old
$ cd /usr/src/linux-2.6.21.7
$ make modules
$ make modules_install
```

Eine Anmerkung: Der letzte Befehl startet auch den Befehl **depmod**, der einige Fehlermeldungen hinsichtlich fehlender Symbole ausgibt. Es handelt sich um Bugs der Entwickler, die fraglichen (unkritischen) Module können damit nicht verwendet werden.

Nach der Installation finden sich im Verzeichnis `/lib/modules/2.6.21.7` die erstellten Module. Zur Installation des IO-Warrior-Treibers vgl. 7.5.2, zu weiteren Details zu Kernel und Modulen vgl. auch die bereits genannte Quelle <http://www.puppylinux.com/development/compilekernel.htm>.

## Legacy-Schnittstellen und digitale IOs

### 7.1 Einführung

Moderne Desktop-PCs und Notebooks weisen mittlerweile oft nur noch Schnittstellen zur Benutzerinteraktion und zur Netzwerkkommunikation auf (DVI, VGA, USB, Ethernet). Die altbekannten seriellen oder parallelen Legacy-Schnittstellen<sup>1</sup> sind durch die neuen komfortableren Busschnittstellen fast völlig verdrängt worden.

Anders sieht der Sachverhalt bei eingebetteten Systemen aus. Noch immer spielt die RS-232-Schnittstelle eine wichtige Rolle für die Kommunikation zwischen Baugruppen im industriellen Umfeld, und entsprechend sind auch moderne Produkte noch mit dieser Schnittstelle ausgestattet. Auch die Drucker- bzw. LPT-Schnittstelle<sup>2</sup> findet noch Anwendung, da diese Schnittstelle mehrere Input-Output-Pins mit sehr geringer Latenz bietet – für eine Prozessanbindung unter Echtzeitanforderungen ist dies ein wichtiges Kriterium (vgl. hierzu auch Abschnitt 1.3.8 und Kapitel 12).

Bei Embedded PCs sind darüber hinaus auch sog. General Purpose Input/Output-Schnittstellen (GPIOs) relativ verbreitet. Diese Schnittstellen sind wahlweise als Ein- oder Ausgang zu parametrieren, und auch sie bieten durch den direkten Prozessoranschluss eine sehr kleine Latenz.

Für Anwendungen, die nicht echtzeitkritisch sind, können die genannten und auch weitere Schnittstellen über USB-Zusatzmodule nachgerüstet werden. Bekannt, und auch unter Linux mittlerweile gut unterstützt, sind USB-RS-232-

---

<sup>1</sup> Legacy: engl. für Erbe. Der Name rührt daher, dass diese Schnittstellen noch von der ersten PC-Generation stammen und mittlerweile nur noch aus Kompatibilitätsgründen mitgeführt werden.

<sup>2</sup> Line Printing Terminal, auch Centronics-Schnittstelle, ein Quasistandard aus den 70er Jahren, mittlerweile abgelöst durch den IEEE-1284-Standard.



und USB-LPT-Wandler, typischerweise basierend auf Bausteinen der Firmen Prolific oder FTDI.<sup>3</sup>

Aber auch gepufferte GPIOs, analoge Schnittstellen, Schnittstellen für Inkrementalgeber u. ä. können über USB-Zusatzmodule nachgerüstet werden. Hierzu werden im letzten Abschnitt dieses Kapitels die IO-Warrior-Bausteine der Firma Code Mercenaries vorgestellt.<sup>4</sup>

## 7.2 RS-232

Die RS-232-Schnittstelle stammt noch aus der Zeit der seriellen Modems und Terminals und ist mittlerweile etwas in die Jahre gekommen. Aber auch, wenn diese serielle Schnittstelle im Alltag an Bedeutung verliert, so besitzt sie in Embedded-Systemen noch immer eine große Relevanz. Fast alle bekannten Mikrocontroller verfügen über mindestens eine serielle Schnittstelle und sparen bei der Kommunikation via RS-232 die zusätzlichen Bausteine, die für andere serielle Verbindungen wie CAN oder USB notwendig sind. Die RS-232-Schnittstelle ermöglicht Übertragungsraten bis zu 115 200 bit/s und ist damit mehr als ausreichend schnell zum Debugging und Flashen, zur Ausgabe von Statusmeldungen oder auch für Konfigurationseinstellungen. Der Schnittstelle liegt der EIA-232-Standard zugrunde, in welchem die Eigenschaften wie Zeitverhalten, Spannungspegel, Protokoll und Steckverbinder festgelegt sind. Serielle Schnittstellen finden sich bei vielen elektronischen Geräten wie Multimetern, Wetterstationen oder Relaiskarten und auch Embedded-µC-Boards; bei Standard-Desktop-Rechnern oder Notebooks verschwinden sie langsam. Aber auch bei diesen PCs können die Schnittstellen leicht durch preisgünstige USB-RS-232-Wandler nachgerüstet werden (vgl. Tabelle E.1 im Anhang).

Im vorliegenden Kapitel sollen zuerst die Grundlagen der RS-232-Kommunikation vermittelt werden. Dann wird in Abschnitt 7.2.2 in die Programmierung der seriellen Schnittstelle eingeführt und mit Abschnitt 7.2.3 als praktisches Beispiel die Ansteuerung einer Relaiskarte vorgestellt.

### 7.2.1 Grundlagen der RS-232-Schnittstelle

#### Datenübertragung und Protokoll

Die für die Datenübertragung zuständigen ICs oder Mikrocontroller-Einheiten werden oftmals als UART<sup>5</sup> bezeichnet. Die verwendeten Bausteine sind in der

<sup>3</sup> <http://www.prolific.com> bzw. <http://www.ftdichip.com>. Unter diesen Adressen kann man entsprechend auch hinsichtlich Treibern fündig werden.

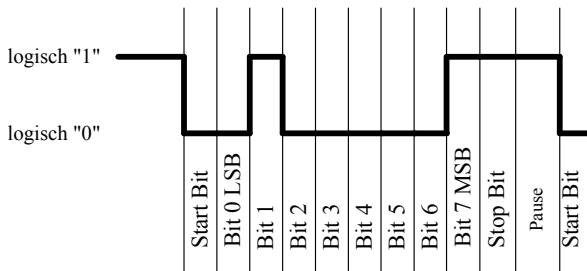
<sup>4</sup> <http://www.codemercs.de>.

<sup>5</sup> Universal Asynchronous Receiver/Transmitter.

Regel kompatibel zum National Semiconductor 8250, dem Urvater aller RS-232-Treiber-ICs. Der 16550D von National Instruments ist ein Nachfolger und wird oft in aktuellen Mainboards eingesetzt.<sup>6</sup> Auf Interna wie Arbeitsweise und Registeraufbau eines solchen Bausteins soll an dieser Stelle nicht eingegangen werden – dieses Wissen wird für die Treiberprogrammierung benötigt, nicht aber für die Verwendung der seriellen Schnittstelle. Für weiterführende Informationen über die direkte Programmierung sei auf die Website von Craig Peacock verwiesen:

<http://www.beyondlogic.org/serial/serial.htm>

Die Datenübertragung erfolgt asynchron, die Dauer zwischen der Übertragung einzelner Bytes kann beliebig lang sein. Innerhalb der Übertragung eines Zeichens müssen allerdings genaue Zeitvorgaben eingehalten werden. Die Synchronisation erfolgt durch ein sogenanntes Startbit (logisch „0“). Danach werden die Daten-Bytes vom Empfänger dem Takt entsprechend eingelesen. Nach der Übertragung eines Zeichens erfolgt eine weitere Synchronisation zwischen Sender und Empfänger durch die Übertragung eines Stopbits (logisch „1“). Abbildung 7.1 zeigt die Übertragung des Zeichens „A“ mit acht Bit je Nutzdaten-Byte, ohne Parität und mit einem Stopbit. Diese 8N1-Einstellung wird standardmäßig bei vielen Geräten verwendet.



**Abb. 7.1.** RS-232-Datenübertragung des Zeichens „A“ mit acht Datenbits, ohne Paritätsbit und mit einem Stopbit (8N1).

Optional kann nach den Nutzdatenbits ein gerades oder ungerades (*even*, *odd*) Paritätsbit übertragen werden, welches der Erkennung von Übertragungsfehlern dient. In der Praxis wird diese Möglichkeit jedoch kaum mehr angewandt. Die nach dem Stopbit folgende Pause darf beliebig lang sein. Durch die getrennten Sende- und Empfangsleitungen ist die Möglichkeit für eine *Full-Duplex*-Übertragung gegeben. Sofern der verwendete Treiber-IC dies unterstützt, ist somit ein simultanes Senden und Empfangen von Da-

<sup>6</sup> Das Datenblatt kann von der Website des Herstellers bezogen werden: <http://www.national.com>. Dort zu finden über eine Suche nach „16550D“.

ten möglich. Neuere Bausteine unterstützen diesen Full-Duplex-Betrieb fast durchgängig, bei älteren Varianten können allerdings Probleme auftreten.

Um Datenverluste zu vermeiden ist die Möglichkeit vorgesehen, den Sender von Empfängerseite aus anzuhalten. Diese Maßnahme war insbesondere in früheren Zeiten wichtig, als die verwendeten Bausteine wenig Pufferspeicher besaßen und damit kaum Daten zwischenspeichern konnten. Man spricht bei diesen Mechanismen auch von *Handshaking* und *Flusskontrolle*. Beim sogenannten Software Handshake werden über die Datenleitungen Signale zum Pausieren des Datenstromes gesendet. Eine Voraussetzung hierfür ist, dass die Signale eindeutig sind und nicht auch in den Nutzdaten auftreten. Beim sogenannten Hardware Handshake steuert der Empfänger den Sendebaustein über separate Leitungen durch seine Ausgänge CTS, DSR und DCD, die empfängerseitig auf RTS und DTR führen (vgl. Abbildung 7.2). Da neuere Bausteine bis zu 64 Byte Datenpuffer besitzen, wird mittlerweile oft vollständig auf eine Flusskontrolle verzichtet.

## Verbindung und Stecker

Von der RS-232-Spezifikation existieren mehrere Varianten, die sich aber nur wenig unterscheiden.

Die verbreitetste Variante ist der RS-232C-Standard, welcher als Spannungspegel für eine logische „1“ einen Bereich zwischen  $-3\text{ V}$  und  $-12\text{ V}$ , für eine logische „0“ den Bereich  $+3\text{ V}$  bis  $+12\text{ V}$  festlegt. Die Signalübertragung erfolgt asymmetrisch und bietet entsprechend relativ wenig Schutz gegenüber Gleichtaktstörungen. Der vergleichsweise hohe Pegel von bis zu  $\pm 12\text{ V}$  lässt in der Praxis dennoch große Leitungslängen von bis zu mehreren hundert Metern zu, obwohl laut Spezifikation bei  $2500\text{ pF}$  Maximalkapazität nur ca.  $15\text{ m}$  Länge möglich wären. Tabelle 7.1 zeigt auf praktischen Versuchen basierende Empfehlungen von Texas Instruments für maximale Kabellängen und Übertragungsraten. Werden längere Strecken oder höhere Übertragungssicherheiten gefordert, so sollte auf einen Standard mit differentieller Übertragung wie RS-422 oder RS-485 ausgewichen werden (vgl. auch den CAN-Bus-Standard, beschrieben in Abschnitt 8.4).

Obwohl die ursprüngliche Norm 25-polige Stecker vorsieht, kommen doch heutzutage aus Platzgründen fast ausschließlich 9-polige DB9- bzw. DE9-Stecker zum Einsatz. Viele der ursprünglich verwendeten Steuerleitungen haben sich im Laufe der Zeit durch verbesserte Übertragungsmechanismen und Software-Protokolle erübrigt. Abbildung 7.2 zeigt Stecker und Belegung einer Steckverbindung, wie sie am Desktop-PC zu finden ist.

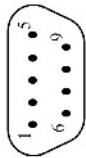
Die einzelnen Pins haben dabei folgende Bedeutung:

- DCD, Data Carrier Detect: Ein Gerät signalisiert dem Computer eingehende Daten.

max. Baudrate [bit/s]	max. Länge [m]
2 400	900
4 800	300
9 600	152
19 200	15
57 600	5
115 200	2

**Tabelle 7.1.** Maximale Kabellängen für RS-232 in Abhängigkeit von der verwendeten Baudrate nach Empfehlungen der Firma Texas Instruments.

Pin	Bedeutung
1	DCD
2	RX
3	TX
4	DTR
5	GND
6	DSR
7	RTS
8	CTS
9	RI



**Tabelle 7.2.** Belegung einer 9-poligen RS-232-Steckverbindung (männliche Ausführung, Stecker, Blick auf die Pins).

- RxD (RX), Receive Data: Leitung für den Empfang von Daten.
- TxD (TX), Transmit Data: Leitung für ausgehende Daten.
- DTR, Data Terminal Ready: Der Host signalisiert dem Gerät, dass er betriebsbereit ist, um es zu aktivieren bzw. zurückzusetzen.
- GND, Ground: Signalmasse. Die Referenz für alle Signalleitungen.
- DSR, Dataset Ready: Signalisiert der Gegenstelle Einsatzbereitschaft (nicht aber Empfangsbereitschaft).
- RTS, Request to Send: Signalisiert der Gegenstelle, dass sie Daten senden soll.
- CTS, Clear to Send: Signalisiert eine Bereitschaft zur Annahme eingehender Daten.
- RI, Ring Indicator: Teilt einen eingehenden Anruf mit (Modem).

Für die Kommunikation zwischen zwei Geräten sind stets die Sendeleitungen (TX) mit den Empfangsleitungen (RX) zu verbinden. Wenn auf ein Hardware Handshake verzichtet werden kann, so ist zusätzlich nur noch eine Masseleitung notwendig. Diese drei Leitungen sollten im Idealfall in einem geschirmten Kabel geführt werden.

Wird eine Hardware-Flusskontrolle verwendet, so sind weiterhin DTR mit DSR+DCD und RTS mit CTS zu verbinden. Dabei gilt die Konvention, dass die Paarung Stecker–Buchse über ein reguläres Verlängerungskabel 1:1 verbunden werden kann, eine Kombination Stecker–Stecker über ein Nullmodemkabel.<sup>7</sup>

## 7.2.2 Ansteuerung und Programmierung

Serielle Schnittstellen können in einem Unix-System auf verschiedenen Ebenen angesprochen werden. Der maschinennahe Zugriff auf einzelne Pins und Register des Schnittstellenwandlers kann via Inline-Makros wie `inb()` und `outb()` erfolgen. Da diese Variante aber eine genaue Kenntnis des verwendeten Schnittstellenbausteins bzw. der internen Register eines Mikrocontrollers voraussetzt, wird üblicherweise eine Ansteuerung auf Dateizugriffsebene verwendet. Hier soll dennoch auch die hardwarenahe Möglichkeit vorgestellt werden, da sich nur mit dieser Art des Zugriffs die Steuersignale als GPIO-Pins zweckentfremden lassen.

Für einen direkten Zugriff auf die Leitungen RTS und DTR muss die Basisadresse des seriellen Ports bekannt sein. In PCs wird die serielle Schnittstelle üblicherweise auf den IO-Speicherbereich gemappt. Ist dies der Fall, so liefert der folgende Aufruf die erforderlichen Adressen:

```
$ cat /proc/ioports | grep serial
02f8-02ff : serial
03f8-03ff : serial
```

Bei einem PC-System wird für die erste serielle Schnittstelle `0x3f8` verwendet, für die zweite Schnittstelle `0x2f8`. Je nach Prozessor wird bei Embedded-Systemen oftmals kein separater Schnittstellenwandler benötigt. Entsprechend entfällt auch das Mapping auf den IO-Speicherbereich, und die Schnittstelle ist im Adressbereich des Prozessors zu finden (vgl. beispielsweise die NSLU2):

```
$ cat /proc/iomem | grep serial
c8000000-c8000fff : serial8250.0
c8000000-c800001f : serial
c8001000-c8001fff : serial8250.0
c8001000-c800101f : serial
```

Ein Blick in das Datenblatt des 16550D zeigt die vorhandenen Register, die relativ zur Basisadresse angesprochen werden (vgl. Abschnitt 7.2.1). Die Steuerleitungen RTS und DTR sind offensichtlich dem *Modem Control Register* und damit der Adresse `base + 4` zugeordnet. Bei einer PC-Schnittstelle mit einer Basisadresse von `0x3f8` wäre ein Setzen der RTS- und DTR-Leitungen mit dem folgenden Listing möglich:

<sup>7</sup> Nullmodem bedeutet: Es wird kein Modem bzw. kein Peripheriegerät angeschlossen, sondern es wird eine Verbindung zwischen zwei PCs hergestellt.

```
#include <sys/io.h>

int main() {
    iopl(3);                      // Zugriff auf IO-Ports, vgl. auch ioperm()
    int base = 0x3f8;             // first register of serial port
    int mcr_reg = inb(base + 4); // read modem control register
    outb(mcr_reg | 0x03, base + 4); // set RTS and DTR pins high
    return 0;
}
```

Eine Anmerkung: Bei Embedded-Systemen sind aus Platzgründen oftmals die Steuerleitungen nicht auf Lötunkte geführt bzw. nicht am Prozessor vorhanden. In diesem Fall muss auf das ohnehin nur noch selten verwendete Hardware Handshaking verzichtet werden.

Die bereits angesprochene elegantere Möglichkeit des Zugriffs auf die serielle Schnittstelle erfolgt wie unter Unix üblich über Dateizugriffsfunktionen. Nur mit dieser Art des Zugriffs kann die Portabilität des Codes gesichert werden. Das folgende Listing zeigt ein Öffnen der Schnittstelle `/dev/ttyS0`<sup>8</sup> mit Ausgabe einer Testnachricht von 21 Zeichen. Das Beispiel ist zu finden unter `<embedded-linux-dir>/examples/legacy/rs232_basic`:

```
#include <stdio.h> // Standard input/output declarations
#include <string.h> // String function declarations
#include <unistd.h> // Unix standard function declarations
#include <fcntl.h> // File control declarations
#include <errno.h> // Error number declarations

int main() {
    int fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);

    if (fd == -1) {
        perror("Unable to open /dev/ttyS0\n");
        return -1;
    }

    if (write(fd, "This is a teststring\n", 21) != 21)
        perror("Write failed\n");

    close(fd);
    return 0;
}
```

Mit den übergebenen Parametern beim Öffnen der seriellen Schnittstelle werden Lese- und Schreibrechte angefordert (`O_RDWR`), die Schnittstelle wird explizit nicht als Konsole verwendet (`O_NOCTTY`) und die Schreib- und Leseoperationen werden als nicht-blockierend durchgeführt (`O_NDELAY`). Die letzte Option hat auf Linux-Systemen den gleichen Effekt wie das Flag `O_NONBLOCK`. Mit der Angabe dieses Flags wird bei lesendem Zugriff nicht auf Daten-Bytes gewartet, sondern im Falle eines leeren Puffers umgehend der Wert 0 zurückgegeben.

<sup>8</sup> Dies ist unter Linux die gebräuchliche Bezeichnung für die erste serielle Schnittstelle; weitere folgen als `/dev/ttyS1` usw.

Mit dem Befehl `cat /dev/ttyS0` ist eine einfache Darstellung der anstehenden Daten möglich. Es sind allerdings mehrere Voraussetzungen zu erfüllen: Es darf kein anderes Terminalprogramm wie z. B. `SerialPort` oder `MiniCom` geöffnet sein, die Schnittstelle muss die richtigen Zugriffsrechte besitzen (vgl. auch Abschnitt A.4.4) und die Sender- und Empfängersysteme müssen korrekt verbunden sein (vgl. Abschnitt 7.2.1).

Eine Anmerkung: Die Kommunikation muss nicht zwangsläufig zwischen zwei Geräten stattfinden. Mit einem Gender-Changer oder einem Nullmodem-Kabel im RS-232-Port kann eine Buchse realisiert werden. Wenn anschließend in dieser bspw. mit einer Büroklammer die Pins 2 und 3 gebrückt werden, so werden ausgegebene Daten auf dieselbe Schnittstelle zurückgesendet. Wird diese Form des Loopbacks verwendet, so ist es wichtig, dass das Terminal die empfangenen Daten nicht automatisch zurücksendet (local echo). Falls die Daten bei der beschriebenen Verkabelung zu schnell über den Bildschirm wandern, so kann das lokale Echo mit dem folgenden Befehl abgeschaltet werden:

```
$ stty -F /dev/ttyS0 -echo
```

## Terminalattribute

Für eine Kommunikation mit seriell angeschlossenen Geräten ist es nicht ausreichend, Daten auf die Schnittstelle zu schreiben oder von dieser einzulesen. Zuerst müssen die in Abschnitt 7.2.1 angesprochenen Übertragungsparameter eingestellt werden. Dies geschieht in einer Struktur vom Typ `termios`, welche in `termios.h` deklariert ist und die Einstellungen für ein sog. Terminal aufnimmt:

```
#include <termios.h> // POSIX terminal control declarations

struct termios {
    tcflag_t c_iflag; // Eingabe-Flag
    tcflag_t c_oflag; // Ausgabe-Flag
    tcflag_t c_cflag; // Kontroll-Flag
    tcflag_t c_lflag; // Lokale Flags
    cc_t      c_cc[NCCS]; // Steuerzeichen
};
```

Innerhalb dieser `tcflag_t`-Strukturen werden die Einstellungen für Geschwindigkeit, Anzahl Daten- und Stopbits und viele mehr hinterlegt. Von den in der gesamten Struktur verfügbaren Flags sind in der Praxis jedoch nur einige wenige relevant. Tabelle 7.3 listet die wichtigsten Flags auf, die für eine Konfiguration der seriellen Schnittstelle benötigt werden.

Mit folgenden, ebenfalls in `termios.h` deklarierten Funktionen werden die in der Struktur `termios` enthaltenen Einstellungen mit einem Terminal ausgetauscht bzw. Daten innerhalb der Struktur manipuliert:

Flag	Enthalten in	Bedeutung
ICRNL	c.iflag	Umwandeln von CR in NL bei der Eingabe
IGNBRK	c.iflag	Break ignorieren
ONLCR	c.oflag	Umwandeln von NL in CR
OPOST	c.oflag	Weitere Bearbeitung der Ausgabe einschalten
CLOCAL	c.cflag	Ausschalten der Modem-Steuerung
CREAD	c.cflag	Aktivieren des Empfängers
CRTSCTS	c.cflag	Einschalten der Hardware-Flusskontrolle
CSIZE	c.cflag	Anzahl Bits für ein Zeichen (CS5 bis CS8)
CSTOPB	c.cflag	Zwei Stop-Bits verwenden (Standard: ein Stopbit)
PARENB	c.cflag	Einschalten der Paritätsprüfung
PARODD	c.cflag	Ungerade Parität (Standard: gerade)
ECHO	c.lflag	Einschalten der ECHO-Funktion
ECHOCTL	c.lflag	Darstellung der Steuerzeichen als Zeichen
ECHOE	c.lflag	Gelöschte Zeichen mit Leerzeichen überschreiben
ECHOK	c.lflag	Ausgabe von NL nach Zeichen löschen
ECHOKE	c.lflag	Zeichen beim Löschen einer Zeile entfernen
ICANON	c.lflag	Zeilenorientierter Eingabemodus
IEXTEN	c.lflag	Einschalten des erweiterten Eingabezeichensatzes
ISIG	c.lflag	Signale (Sonderzeichen) einschalten

**Tabelle 7.3.** Übersicht über die wichtigsten Flags der Struktur `termios`.

```
int tcgetattr(int fd, struct termios *termios_p);
```

Diese Funktion speichert die aktuell festgelegten Terminalparameter in einer Struktur vom Typ `termios`, auf welche `termios_p` zeigt. `fd` gibt an, von welchem Terminal die Daten zu beziehen sind. Im Fehlerfall wird `-1` zurückgeliefert, sonst `0`.

```
int tcsetattr(int fd, int optional_actions, const struct
termios *termios_p);
```

Mit dieser Funktion werden die Parameter für ein Terminal `fd` gesetzt. Die Parameter werden aus einer `termios`-Struktur entnommen, auf welche `termios_p` zeigt. Die Angabe `optional_actions` bestimmt, wann Änderungen aktiv werden:

- `TCSANOW` – Änderungen werden sofort aktiv.
- `TCSADRAIN` – Änderungen werden aktiv, sobald alle Zeichen aus dem Puffer ausgegeben wurden.
- `TCSAFLUSH` – wie `TCSADRAIN`, jedoch werden alle noch im Puffer enthaltenen Zeichen verworfen.

Im Fehlerfall wird `-1` zurückgeliefert, sonst `0`.

```
int cfsetispeed(struct termios *termios_p, speed_t speed);
```

Diese Funktion legt die Baudrate der Eingangsdaten in der Struktur



fest, auf die `termios_p` zeigt. Zu beachten ist, dass die Übernahme der neuen Werte nicht automatisch erfolgt, sondern gesondert mittels `tcsetattr()` durchgeführt werden muss. Als Geschwindigkeit werden die in `termios.h` definierten Makros verwendet (B50,..., B9600,..., B115200). Im Fehlerfall wird `-1` zurückgeliefert, sonst `0`. Zur Festlegung der Geschwindigkeit für Eingangs- und Ausgangsdaten stehen `cfsetispeed()` und `cfsetospeed()` zur Verfügung. Die Funktionen `cfgetispeed()` und `cfgetospeed()` geben entsprechend die eingestellte Geschwindigkeit zurück.

Wurden mit obigen Befehlen die Einstellungen für eine serielle Schnittstelle vorgenommen, so kann das Ergebnis mit dem Programm `stty`<sup>9</sup> überprüft werden, mit welchem u. U. bereits die `echo`-Funktion des Terminals abgeschaltet wurde:

```
$ stty -F /dev/ttyS0
speed 115200 baud; line = 0;
kill = ^X; min = 1; time = 0;
-brkint -icrnl -imaxbel
-opost -onlcr
-isig -icanon -iexten -echo -echoe -echok -echoctl -echoke
```

Ein vorangestelltes „-“ bedeutet eine Negation des jeweiligen Flags. Anzumerken ist, dass sich die Schnittstellenparameter hiermit auch an der Kommandozeile setzen lassen. Die Argumente sind mit den Flags aus Tabelle 7.3 nahezu identisch.

Da je nach Verbindung üblicherweise nur die Baudrate, die Anzahl der Stopbits und Datenbits sowie die Parität variiert, ist es sinnvoll, die sehr umfangreichen Einstellungen zu kapseln und eine übersichtliche API zu verwenden. Im folgenden Abschnitt übernimmt diese Aufgabe die Klasse `SerialPort`.

## Die Klasse `SerialPort`

Die Klasse `SerialPort` (vgl. `<embedded-linux-dir>/src/tools/`) ist sehr einfach aufgebaut und schöpft entsprechend auch nicht alle Konfigurationsmöglichkeiten der seriellen Schnittstelle aus. Ziel ist vielmehr, hiermit dem Anwender eine API für die typischen Konfigurationen zur Verfügung zu stellen. Als Membervariablen werden lediglich der File Descriptor und eine Struktur vom Typ `termios` benötigt, um die ursprünglichen Einstellungen zwischenspeichern:

```
int m_fd; // file descriptor
struct termios m_options_old; // previous options for serial port
```

Die Konfiguration der Schnittstelle wird bereits im Konstruktor der Klasse festgelegt, sodass lediglich zwei weitere Methoden für die Datenkommunikation notwendig sind. Die API sieht damit folgendermaßen aus:

<sup>9</sup> Programm zum Setzen der Parameter für eine Teletype-Schnittstelle (TTY).

```
SerialPort(char* device, speed_t speed, int parity, int
    databits, int stopbits, bool block);
```

Die Funktion erzeugt ein Objekt vom Typ `SerialPort` und öffnet eine serielle Schnittstelle mit Gerätenamen `device`.

In Abhängigkeit von der Variablen `block` geschieht dies für blockierende oder nicht-blockierende Lese- bzw. Schreibzugriffe. Die aktuell eingestellte Konfiguration wird zwischengespeichert. `speed` gibt die Baudrate für Ein- und Ausgabe an und kann Ganzzahlwerte entsprechend den in `termios.h` hinterlegten Makros annehmen. Die Parität wird über den Wert `parity` festgelegt. Hierbei entspricht (0,1,2) den Einstellungen (`no`,`odd`,`even`). Die Anzahl der Datenbits und Stopbits werden in `databits` und `stopbits` festgelegt.

```
int writeData(unsigned char* buf, int num);
```

Die Funktion schreibt `num` Daten-Bytes, beginnend von Speicherstelle `buf`, auf die serielle Schnittstelle. Im Erfolgsfall wird die Anzahl übermittelter Bytes zurückgeliefert, sonst `-1`.

```
int readData(unsigned char* buf, int num);
```

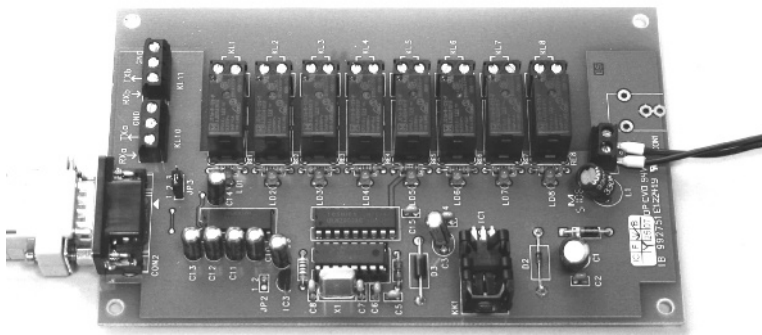
Die Funktion liest `num` Daten-Bytes von der seriellen Schnittstelle und legt diese an der Speicherstelle `buf` ab. Im Erfolgsfall wird die Anzahl empfangener Bytes zurückgeliefert, sonst `-1`.

Das Beispiel `<embedded-linux-dir>/examples/legacy/rs232.cpp` zeigt die Verwendung der C++-Schnittstelle. Mit nur wenigen Codezeilen ist es damit möglich, die serielle Schnittstelle zu konfigurieren und Daten zu versenden. Das Ergebnis kann durch Eingabe von `cat /dev/ttyS0` in einem zweiten Konsolenfenster nachvollzogen werden (Anm.: hierzu ist wieder die Überbrückung der Pins 2 und 3 notwendig).

### 7.2.3 Ansteuerung einer seriellen Relaiskarte

Als Beispiel für die serielle Kommunikation wird in diesem Abschnitt eine Relaiskarte der Fa. Conrad Electronic vorgestellt (vgl. Abbildung 7.2 und Tabelle E.1 im Anhang). Die Karte trägt acht Relais, die seriell angesteuert werden können. Durch eine entsprechende Konfiguration ist es möglich, bis zu 255 dieser Relaisplatinen zu kaskadieren. Die Schaltspannung der Relais beträgt maximal 24 V. Das Schalten von 230 V ist nicht zulässig.

Die Platine wird über ein Nullmodemkabel mit der Gegenstelle verbunden und kann über einfache Kommandos angesprochen werden (vgl. Tabelle 7.4). Als Übertragungsparameter werden 19 200 Baud, 8 Datenbits, keine Parität und 1 Stopbit verwendet. Auf eine Flussteuerung wird verzichtet. Die für eine Kommunikation notwendige Adresse lautet bei Verwendung einer einzelnen Platine „1“, bei mehreren Platinen ergibt sie sich aus der Position in der Kaskade.



**Abb. 7.2.** Serielle Relaiskarte der Firma Conrad mit acht Relais (vgl. Tabelle E.1).

Kommando	Bedeutung	Antwortkommando
0	No Operation	255
1	Initialisierung	254
2	Abfrage der Schaltzustände	253
3	Setzen der Schaltzustände	252
4	Abfrage der Optionen	251
5	Setzen der Optionen	250

**Tabelle 7.4.** Auflistung der Kommandos.

Grundsätzlich werden zwischen Karte und Gegenstelle immer jeweils vier Bytes kommuniziert. Als Antwort erfolgt von der Relaisplatine eine Nachricht mit Kommando (255-Sendekommando) oder eine Fehlernachricht 255. Der Rahmen ist dabei folgendermaßen aufgebaut:

	1. Byte		2. Byte		3. Byte		4. Byte	
	Kommando		Adresse		Daten		Prüfsumme	

Werden abgesehen von den Kommandos 3 und 5 keine Daten übermittelt, so ist der Wert für das dritte Byte beliebig. Die Prüfsumme im vierten Byte wird durch exklusive Oder-Verknüpfung (XOR) der Bytes 1 bis 3 gebildet. Bei nicht korrekter Prüfsumme erfolgt eine Fehlermeldung (Kommando 255). Bei einer Initialisierung mittels Kommando 1 wird die Adresse der ersten angeschlossenen Platine gesetzt. Nachfolgende Platinen erhalten jeweils eine um 1 erhöhte Adresse. Nach der Initialisierung erfolgt durch jede einzelne Platine eine Bestätigung in Form des Kommandos 254.

Die Befehle 2 und 3 setzen bzw. lesen den aktuellen Status der Relais. Das niederwertigste Bit 0 im Daten-Byte entspricht dabei Relais 1 (bei einer Benennung von 1...8). Über die Kommandos 4 und 5 werden Broadcast-Optionen

abgefragt bzw. gesetzt, die bei Verwendung mehrerer Platinen relevant werden. Das mitgelieferte Datenblatt liefert hierzu weitere Informationen.<sup>10</sup>

Die im folgenden Abschnitt vorgestellte Klasse **RelaisBoard** repräsentiert Objekte dieses Typs und erlaubt auch ohne Kenntnis des spezifischen Protokolls eine einfache Verwendung der Relaisplatine.

## Die Klasse RelaisBoard

Die Klasse erlaubt den Zugriff auf einzelne bzw. nicht kaskadierte Relaiskarten. Erfordert die Applikation die Verwendung mehrerer Relaiskarten, so ist die Klasse entsprechend zu erweitern.

Damit die Kartenparameter nicht vor jedem Schreibvorgang bestimmt werden müssen, wird der aktuelle Zustand der Karte hinterlegt. Die von der Karte empfangenen Werte für Kommando, Adresse und Daten werden zur Weiterverarbeitung ebenfalls im Objekt abgespeichert. Weiterhin ist als Membervariable eine Referenz auf das Objekt der seriellen Schnittstelle notwendig:

```
SerialPort& m_serial;           // reference to serial port
unsigned char m_state;          // relais states [0-255]
unsigned char m_cmd, m_addr, m_data; // response bytes from card
```

Für den Anwender steht folgende API zur Verfügung:

```
RelaisBoard(SerialPort& serial);
```

Die Funktion erzeugt ein Objekt vom Typ **RelaisBoard** und initialisiert eine an **serial** angeschlossene Relaiskarte.

```
int setRelaisOn(unsigned char rel);
```

Die Funktion aktiviert die in **rel** angegebenen Relais. Bei erfolgreicher Durchführung wird 0 zurückgegeben, sonst -1.

```
int setRelaisOff(unsigned char rel);
```

Die Funktion deaktiviert die in **rel** angegebenen Relais. Bei erfolgreicher Durchführung wird 0 zurückgegeben, sonst -1.

```
int getStatus();
```

Die Funktion liefert im niederwertigsten Byte ein Bitfeld mit den aktuellen Schaltzuständen zurück. Diese werden zuvor von der Karte selbst gelesen. Bei Auftreten eines Fehlers wird -1 zurückgegeben.

Die Hilfsfunktion **sndCmd()** sendet ein Kommando mit zugehörigem Datenwert an eine bestimmte Kartenadresse. Im Anschluss daran liest sie die vier Antwort-Bytes mit der Funktion **recvCmd()** aus. Das Beispiel `<embedded-linux-dir>/examples/legacy/rs232.relaisboard` zeigt die Verwendung der Klassen **SerialPort** und **RelaisBoard**. In Abschnitt 11.6

<sup>10</sup> Vgl. `<embedded-linux-dir>/datasheets/relaisplatine-8-fach.pdf`.

wird im Kapitel *Gerätetreiber und Kernelmodule* als praktische Anwendung die Ansteuerung einer Relaiskarte in ein Kernelmodul integriert.

Eine Anmerkung: Falls zur Ansteuerung der seriellen Relaiskarte am PC ein USB-RS-232-Adapter mit PL2303-Chipsatz zum Einsatz kommt, so können hierbei Probleme auftreten (zum Produkt vgl. Tabelle E.1 im Anhang).

Bei unseren Versuchen hat zwar das Setzen der Relaiszustände immer fehlerfrei funktioniert, der gelesene Zustand hat allerdings häufiger nicht mit dem tatsächlichen Zustand übereingestimmt. Hier ist gegebenenfalls auf das Rücklesen dieses Wertes zu verzichten. Alternativ kann auf Adapter mit dem ebenfalls relativ verbreiteten FTDI-Chipsatz ausgewichen werden.<sup>11</sup>

### 7.3 Centronics und IEEE 1284

Die parallele Centronics-Schnittstelle wurde von der gleichnamigen Firma entwickelt und konnte sich rasch als Quasistandard für die Druckeranbindung durchsetzen. Die sonstige Nutzung der Schnittstelle war relativ eingeschränkt, da diese nur eine unidirektionale Übertragung ermöglicht. Daher wurde im Jahre 1994 mit der IEEE 1284-Norm eine Erweiterung hinsichtlich der bidirektionalen Übertragung verabschiedet. Schnell wurde dieses Interface auch für Scanner, ZIP-Laufwerke und andere Peripherie-Geräte genutzt. Auch wenn für diese Gerätearten mittlerweile meist USB verwendet wird, so ist die parallele Schnittstelle noch immer gerade im Embedded-Bereich besonders interessant, da sie mehrere einfach und schnell ansteuerbare digitale Input/Output-Pins bietet.

Die Schnittstelle wird auch Line Printing Terminal (LPT, genauer: LPT1, LPT2...) <sup>12</sup> genannt. Die Pinbelegung ist in Tabelle 7.5 aufgeführt.

Die maximale Übertragungsrate beträgt 2 Mbit/s, die maximale Leitungslänge ca. 10 m. Die Eingänge sind über interne Widerstände von 2 k $\Omega$ –5 k $\Omega$  auf 5 V gelegt (Pull-Up), um definierte Pegel festzulegen. Die Basisadresse des Datenregisters lautet 3BC<sub>hex</sub>, 378<sub>hex</sub> oder 278<sub>hex</sub>. Das Statusregister liegt an Basisadresse + 1, das Steuerregister an Basisadresse + 2.

Der Centronics- bzw. der IEEE 1284-Standard definieren über dieser physikalischen Schicht noch eine weitere logische Schicht. Zu dieser Festlegung vgl. [Wikipedia 08, IEEE\_1284]. Bei diesem Highlevel-Zugriff kann über Pin 2–9 bei der Datenausgabe parallel ein Byte gesendet werden. Pin 1 (Strobe) ist high, wenn keine Daten für das Peripheriegerät bereitstehen. Wenn Daten gesendet werden sollen, so wird dieser Ausgang für ca. 5  $\mu$ s auf low gezogen. Über Pin 10 (Acknowledge, ACK) wird die Übertragung im Erfolgsfalle bestätigt.

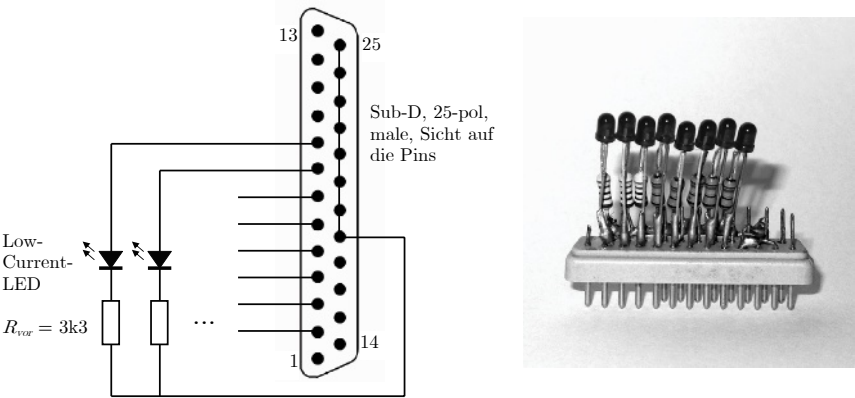
<sup>11</sup> Website des Herstellers: <http://www.ftdichip.com>.

<sup>12</sup> Entsprechend unter Linux: Devices /dev/lp1, /dev/lp2 ...

Pin	Typ	Funktion
1	Ausgang	Steuerleitung 0 (invertiert)
2	Ausgang	Datenleitung 0
3	Ausgang	Datenleitung 1
4	Ausgang	Datenleitung 2
5	Ausgang	Datenleitung 3
6	Ausgang	Datenleitung 4
7	Ausgang	Datenleitung 5
8	Ausgang	Datenleitung 6
9	Ausgang	Datenleitung 7
10	Eingang	Statusleitung 6
11	Eingang	Statusleitung 7 (invertiert)
12	Eingang	Statusleitung 5
13	Eingang	Statusleitung 4
14	Ausgang	Steuerleitung 1 (invertiert)
15	Eingang	Statusleitung 3
16	Ausgang	Steuerleitung 2
17	Ausgang	Steuerleitung 3 (invertiert)
18-25	–	GND

**Tabelle 7.5.** Pinbelegung der parallelen Schnittstelle (vgl. auch [Gräfe 05]).

Das Stromliefervermögen der Ausgänge variiert je nach verwendetem Interface-Baustein und liegt bei insgesamt ca. 12 mA. Weiterhin wird zwischen dem Source- und Sink-Stromliefervermögen gegen 5 V bzw. GND unterschieden – mit einer Annahme von maximal 1 mA für Source und Sink liegt man meist auf der sicheren Seite.



**Abb. 7.3.** Testschaltung für den Parallelport.

Für den Umgang mit der parallelen Schnittstelle ist eine kleine Testschaltung hilfreich. Diese kann z. B. aussehen, wie in Abbildung 7.3 gezeigt. Anzumerken ist, dass bei uns im Labor mit dieser Schaltung bisher noch nie ein Schaden an den Schnittstellen entstanden ist – generell auszuschließen ist dies aber nicht.

Für den Zugriff aus eigenen Programmen heraus auf die Schnittstelle existieren mehrere Möglichkeiten. Die einfachste Art des Zugriffs erfolgt über die genannten Devices `/dev/lp<n>`. Hierbei folgt die Ansteuerung dem Druckerprotokoll, realisiert durch das genannte Data-Bit, das Strobe-Bit und das Acknowledge-Bit. Initialisierung, Ein- und Ausgabe erfolgen durch Öffnen, Lesen und Schreiben des Devices. Da diese Art der Ansteuerung langsam und nur wenig flexibel ist, soll im Weiteren der direkte Port-Zugriff vorgestellt werden (vgl. auch Abschnitte 11.6 und 12.5).

Wenn an den Parallelport eine Schaltung gem. Abbildung 7.3 angeschlossen wird, so realisiert das folgende Listing einen LED-Binärzähler.

```
// Ende mit Ctrl+C
#include <stdio.h>
#include <unistd.h>
#include <sys/io.h> // ersatzweise: asm/io.h fuer aeltere Linuxe

int main()
{
    unsigned char c = 0;
    iopl(3);          // Zugriff auf IO-Ports, vgl. auch ioperm()
    while (1)
    {
        outb(c, 0x378); // zur Auswahl steht: 0x278, 0x378, 0x3bc
        usleep(100000L); // n Mikrosekunden warten
        c++;
    }
    return 0;
}
```

Hierin regelt die Funktion `iopl()` den Hardware-Zugriff über das mitgegebene Level (0 für keine Erlaubnis, 3 für vollen Zugriff). Ersatzweise kann hierzu auch die Funktion `ioperm()` verwendet werden, welcher ein bestimmter freizugebender Adressbereich mitgegeben wird:

```
int ioperm(unsigned long begin, unsigned long num, int val);
```

Die Variable `val` kann die Zustände 1 (Zugriff erlauben) und 0 (Zugriff sperren) annehmen [Gräfe 05]. Die Basisadresse der parallelen Schnittstelle im Listing ist u. U. anzupassen; sie kann folgendermaßen ermittelt werden:

```
cat /proc/ioports | grep parport
```

Für einen Compiler-Durchlauf ist dem gcc die Option `-O` oder `-O2` mitzugeben, um Inline-Makros wie `outb()` verfügbar zu machen. Weiterhin muss entweder das generierte Programm mittels `sudo` aufgerufen oder aber dessen Programmrechte auf root gesetzt werden. Zum abschließend erforderlichen `chmod` vgl. Anhang A. Die Übersetzung des Quelltextes und der Aufruf sind entsprechend wie folgt einzugeben:

```
gcc -O LED-zaehler.c -o LED-zaehler
su
chown root:root LED-zaehler
chmod a+s LED-zaehler
exit
./LED-zaehler
```

Zu weiteren Details bzgl. den I/O-Port-Makros vgl. folgende URLs:

<http://linux.die.net/man/2/ioperm>  
<http://linux.die.net/man/2/iopl>  
[http://linux.die.net/man/2/outw\\_p](http://linux.die.net/man/2/outw_p)  
<http://linux.die.net/man/2/inb>  
 ...

Eine Anmerkung zur Basisadresse: Falls die Adresse nicht bekannt ist bzw. nicht vom Anwender im Programm eingestellt wurde, so kann testhalber eine 0 auf alle in Frage kommenden Adressen geschrieben und anschließend wieder gelesen werden. Gibt der Lesevorgang 255 zurück, so ist an der getesteten Adresse keine Schnittstelle vorhanden [Gräfe 05, Kap. 9].

Weiterhin kann unter Linux für einen komfortablen Low-Level-Zugriff auch das *ppdev*-Modul oder die *libieee1284*-Bibliothek verwendet werden. Vgl. hierzu die Include-Datei `/usr/include/linux/ppdev.h` und folgende URLs:

<http://people.redhat.com/twaugh/parport/html/x623.html>  
<http://as6edriver.sourceforge.net/hacking.html>  
<http://sourceforge.net/projects/libieee1284>  
<http://www.ilive4unix.net/doku.php/projects/libmddriver/libieee1284-test.c>

## 7.4 General Purpose Input/Output (GPIO)

GPIOs sind Schnittstellen eines Prozessors, die als digitale Eingänge oder Ausgänge konfiguriert werden können. Je nach Bauart können diese Schnittstellen so eingestellt werden, dass sie einen Interrupt auslösen oder auch direkten Speicherzugriff ermöglichen (Direct Memory Access, DMA). Der Spannungsbereich der GPIOs entspricht meist der Betriebsspannung des Prozessors (z. B. 0 V–3,3 V oder 0 V–5 V). Üblicherweise sind sie in Achtergruppen als Port arrangiert.

Wenn ein entsprechender Treiber vorliegt, können die GPIOs einfach und komfortabel als Device angesprochen werden. An der Konsole oder in Skripten geschieht dies über die `proc`-Erweiterung, in eigenen Quelltexten über die `ioctl()`-Funktion. Diese zwei Varianten wurden bereits in den Abschnitten 5.5.1 und 5.5.2 am Beispiel des Alekto vorgestellt.



Die dritte Möglichkeit ist der systemnahe Zugriff über Port-Makros. Die Vorgehensweise ist ähnlich dem Zugriff auf die parallele Schnittstelle (vgl. Abschnitt 7.3). Als Beispiel sei der etwas exotische MicroClient SX-PC angeführt, von welchem Derivate existieren, die eine GPIO-Schnittstelle an der Front aufweisen. Die Schnittstelle wird von den für den SX empfohlenen Betriebssystemen Puppy Linux und Windows CE nicht unterstützt. Über eine Suche beim Chipsatz-Hersteller gelangt man aber schließlich auch zu einer Dokumentation der GPIOs. Weiterhin hat der Hersteller freundlicherweise auch ein kurzes C-Beispiel für Linux beigelegt:

<http://www.vortex86sx.com>

[ftp://download@ftp.dmp.com.tw/vortex86sx/SX\\_Using\\_GPIO.pdf](ftp://download@ftp.dmp.com.tw/vortex86sx/SX_Using_GPIO.pdf)

```
#include <stdio.h>
#include <stdio.h>
#include <sys/io.h>
#define outportb(a,b) outb(b,a)
#define inportb(a) inb(a)

void main(void)
{
    iopl(3);
    // set GPIO port0[7-0] as input mode
    outportb(0x98, 0x00);
    // read data from GPIO port0
    inportb(0x78);
    // set GPIO port1[7-0] as output mode
    outportb(0x99, 0xff);
    // write data to GPIO port1
    outportb(0x79, 0x55);
    // set GPIO port2[7-4] as output and [3-0] as input
    outportb(0x9a, 0xf0);
    // write data to GPIO port2[7-4], the low nibble (0x0a) will be ignored
    outportb(0x7a, 0x5a);
    // read data from port2[3-0]
    unsigned char c = inportb(0x7a) & 0x0f;
    // if GPIO port3 is free, those codes can work
    // set GPIO port3[7-2] as output and [1-0] as input
    outportb(0x9b, 0xfc);
    // write data to GPIO port2[7-2], the bit 1-0 will be ignored
    outportb(0x7b, 0xa5);
    // read data from port3[1-0]
    unsigned char c = inportb(0x7b) & 0x03;
    // if GPIO port4 is free, those codes can work
    // set GPIO port4[7,5,3,1] as output and port4[6,4,2,0] as input
    outportb(0x9c, 0xaa);
    // write data to GPIO port4[7,5,3,1], the bit 6,4,2 and will be ignored
    outportb(0x7c, 0xff);
    // read data from port4[6,4,2,0]
    unsigned char c = inportb(0x7c) & 0xaa;
}
```

Zur Compiler-Aufrufkonvention und zum notwendigen `chown` und `chmod` vgl. Abschnitt 7.3. Abschließend eine Anmerkung zum GPIO-Zugriff auf der NSLU2: Der dort verwendete IXP420-Prozessor besitzt immerhin 16 GPIO-

Pins<sup>13</sup>, die bei Linux-Anwendern Begehrlichkeiten wecken. Leider sind aber die meisten GPIOs bereits für interne Funktionen wie Power Off, Power Button, PCI Reset uvm. belegt (vgl. hierzu auch Tabelle 2.3).

Es können aber zumindest jene GPIOs, die zur Ansteuerung der Status-Leuchtdioden zuständig sind, relativ einfach zweckentfremdet werden. Die LEDs können bei einer hochohmigen Beschaltung weiter auf dem Board verbleiben (vgl. auch [ct 08, Heft 10, S. 202 ff.]). Die Ansteuerung erfolgt wie bereits beschrieben via `ioctl()`-Befehlen. Details sind folgender Header-Datei zu entnehmen:

```
/usr/include/asm/arch-ixp4xx/nslu2.h
```

## 7.5 Schnittstellenerweiterung über IO-Warrior

### 7.5.1 IO-Warrior-Bausteine

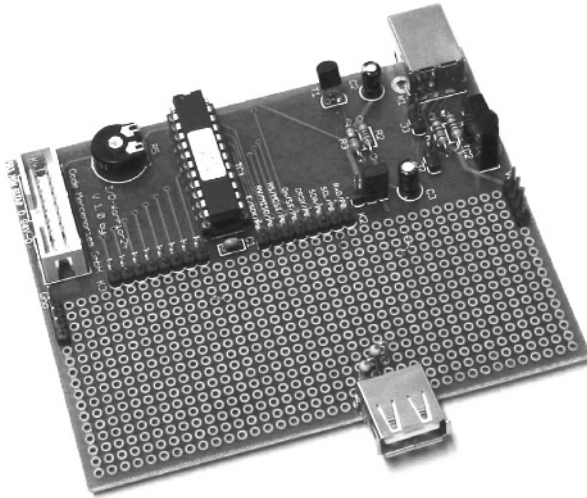
Sind an einem Embedded-Board keine freien GPIO-Pins verfügbar oder sind die Pins nur schwer zugänglich, so können auch mittels eines IO-Warrior-Bausteins der Fa. Code Mercenaries [Codemercs 08] GPIOs nachgerüstet werden. Die IO-Warrior-Bausteine kommunizieren als HID-Geräte<sup>14</sup> mit dem Host-Rechner, ähnlich einer Tastatur oder einer Maus. Auf Windows- und Mac OS-Systemen erfolgt die Anbindung über vorhandene Systemtreiber. Für Linux steht ein Open-Source-Treiber zur Verfügung.

Es existieren verschiedene Warrior-Varianten, welche alle als HID-Bausteine angesprochen werden. IO-Warrior-Bausteine stellen als universelle IO-Controller Schnittstellen wie SPI, I<sup>2</sup>C, General Purpose IOs und RC5-Infrarotempfänger zur Verfügung. Darüber hinaus bieten sie Unterstützung zur Ansteuerung von alphanumerischen LCDs und Matrix-LEDs. Abbildung 7.4 zeigt das IO-Warrior24-Starterkit.

Auf der Hersteller-Website werden auch andere Varianten der Warrior-Bausteine vorgestellt. So dekodiert und zählt bspw. der SpinWarrior24-A3 Signale eines Quadraturencoders und der JoyWarrior24-F8 misst Beschleunigungen in mehreren Achsen. Da der IO-Warrior in diesem Buch primär der universellen I<sup>2</sup>C-Erweiterung dient, reicht der *IOW24*-Baustein völlig aus. Werden je nach Anwendung zusätzliche IO-Leitungen benötigt, so kann auf die Varianten *IOW40* oder *IOW56* zurückgegriffen werden.

<sup>13</sup> Genauer: Balls, da es sich um ein Ball Grid Array (BGA) handelt. Entsprechend kann sich auch das Herausführen einer Schnittstelle als schwierig erweisen.

<sup>14</sup> Human Interface Device: Eine Klasse von USB-Geräten, die direkt mit dem Anwender interagieren.



**Abb. 7.4.** IO-Warrior24-Starterkit (100×80 mm<sup>2</sup>) mit Anschluss für LCD, IR-Receiver, GPIO und I<sup>2</sup>C über Stiftleisten. Die USB-A-Buchse mit Pull-Up-Widerständen wurde als elektromechanische I<sup>2</sup>C-Schnittstelle nachträglich angebracht.

Für die Kommunikation mit den Bausteinen der Serien *IOW24* und *IOW40* werden vom Hersteller Laufzeitbibliotheken für Windows und Linux zur Verfügung gestellt. Die Linux-Bibliothek `libiowkit.so` ist aktuell in der Version 1.5 verfügbar. Die API ist für beide Bibliotheken identisch und in `<embedded-linux-dir>/datasheets/IO-Warrior Library.pdf` erklärt. Die Laufzeitbibliothek greift bei Linux auf Kernelfunktionen zu, welche im Treibermodul `iowarrior.ko` enthalten sind. Der nachfolgende Abschnitt 7.5.2 widmet sich der Treiberinstallation und dem Übersetzen der IOW-Bibliothek.

### 7.5.2 Installation der IO-Warrior-Treiber unter Debian

Die IO-Warrior-Treiber werden in Form eines Kernelmoduls in das System integriert. Laut Herstellerangaben soll der Treiber ab Linuxversion 2.6.21 im Kernel enthalten sein. Wenn ohnehin ein eigener Kernel übersetzt wird, so sollte diese Möglichkeit zunächst überprüft werden – damit erübrigen sich unter Umständen die folgenden Schritte. Für eine Übersetzung des Modules auf dem Zielsystem ist das Vorhandensein der Kernel-Header-Dateien Voraussetzung. Zu genaueren Ausführungen über Programmierung und Aufbau von Kernelmodulen sei an dieser Stelle auf Kapitel 11 verwiesen. Die Installation der Linux-Headers geschieht unter Debian durch folgenden Aufruf:<sup>15</sup>

<sup>15</sup> Für Puppy ist diese Prozedur in Abschnitt 6.4 beschrieben.

```
$ sudo apt-get install linux-headers-<versionsnummer>
```

Alternativ können natürlich auch die Kernelquellen installiert werden. Die Versionsnummer (mit Endung) des auf dem Zielsystem laufenden Kernels wird durch einen Aufruf von `uname` bestimmt:

```
$ uname -r
2.6.18-4-ixp4xx
```

Abweichungen bei der Installation für unterschiedliche Debian-Distributionen sind, bis auf kleine Details, lediglich hinsichtlich der Linux-Header zu erwarten. OpenWrt bildet hier mit der Integration in das eigene Build-System eine Ausnahme. Die spezielle Vorgehensweise wird in Kapitel 3 ausführlich beschrieben.

Die IO-Warrior-Treiber werden vom Hersteller [Codemercs 08] online bereitgestellt.<sup>16</sup> Nach dem Herunterladen und Entpacken liegt die aktuelle Treiberversion in Verzeichnis `LinuxSDK/Kernel_2.6/iowkit 1.5`. Unsinnigerweise wurde hier ein Leerzeichen in den Pfadnamen `iowkit 1.5` eingebaut. Dieses verursacht beim Übersetzen des Treibers Probleme und sollte vorher entfernt werden.

Zunächst wird das Treibermodul übersetzt und installiert. Abhängig von der verwendeten Kernelversion werden die Quellen aus der Datei `iowarrior-module-2.6.tar.gz` (Kernelversion kleiner 2.6.20) oder `iowarrior-module-2.6.20.tar.gz` (Kernelversion 2.6.20 oder höher) entpackt. Das Kernelmodul kann im neuen Verzeichnis direkt mit `make` übersetzt werden.<sup>17</sup>

```
$ cd iowarrior-module-2.6
$ make
$ sudo make install
```

Nach dem Übersetzungsprozess sollte die Datei `iowarrior.ko` erstellt und in das Zielverzeichnis kopiert worden sein (die Installation muss mit Root-Rechten ausgeführt werden). Weiterhin wurden Abhängigkeiten mit `depmod` untersucht und das Modul über `modprobe` geladen – der Befehl `lsmod` liefert den Beweis. Mit dem mitgelieferten Skript können nun die Einsprungspunkte in den Treiber bzw. die Gerätedateien installiert werden. Falls `udev` vorhanden ist, geschieht dies über folgenden Befehl:<sup>18</sup>

```
$ sudo ./make_iow_devices
```

<sup>16</sup> Enthalten in der Rubrik *Downloads/Software* als Datei *IO-Warrior SDK Linux.zip*.

<sup>17</sup> Unter Puppy muss vor dem Übersetzen noch ein symbolischer Link erstellt werden: `ln -s /lib/modules/2.6.21.7/build /usr/src/linux-2.6.21.7`. Das Zielverzeichnis für die Installation `/lib/modules/2.6.21.7/kernel/drivers/usb/misc/` ist hier ebenfalls von Hand zu erstellen.

<sup>18</sup> Die Datei `make_iow_devices` scheint standardmäßig nicht ausführbar zu sein. Dies ist einzustellen via `chmod a+x make_iow_devices`.

Die dynamische Erzeugung der Gerätedateien mit `udev` ist die elegantere Variante. Sollte das Programm nicht verfügbar sein (z.B. bei Puppy), dann können statische Gerätedateien über die Option `--force-static-nodes` auch herkömmlich mit `mknod` erzeugt werden. Wird nun ein IO-Warrior-Modul an den USB-Port angeschlossen, so sollten bei dynamischer Erzeugung zwei Geräte `/dev/iowarrior0` und `/dev/iowarrior1` zur Kommunikation mit der IO-Schnittstelle bzw. für die Spezialfunktionen erzeugt werden.

Es ist zu beachten, dass je nach verwendeter Spezialfunktion manche IOs belegt und daher über Gerät 0 nicht mehr erreichbar sind. Die Zuordnung der beiden Schnittstellen zu Gerätenamen kann in separaten Threads geschehen. Dies bedeutet, dass eine durchgängige Nummerierung nicht gewährleistet ist. Wenn mehrere Module angeschlossen sind, so müssen die Geräte mit Endnummern 0 und 1 also keineswegs zum selben IO-Warrior gehören. Letztendlich ist eine genaue Identifikation von Baustein und Funktionsregister nur über die Seriennummer des Gerätes zu erreichen. Beispiele zur Verwendung der I<sup>2</sup>C-Schnittstelle der IO-Warrior-Bausteine werden in Abschnitt 8.3.3 vorgestellt.

Auch wenn alle bereitgestellten Funktionen ebenfalls über die beiden Gerätedateien realisiert werden können, so bietet die mitgelieferte Bibliothek `libiowkit-1.5.0` doch eine etwas komfortablere Schnittstelle (dies gilt leider nicht für die I<sup>2</sup>C-Funktionen). Zum Übersetzen der Bibliothek ist in das Verzeichnis `libiowkit-1.5.0` zu wechseln und dort die Standardprozedur für die Erstellung und Einbindung neuer Software durchzuführen:

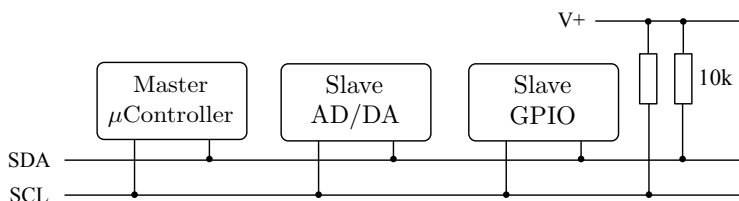
```
$ ./configure
$ make
$ sudo make install
```

Beispiele für die Verwendung der Bibliothek finden sich im Handbuch *IO-Warrior Dynamic Library* in `LinuxSDK.zip`.

## Der Inter-IC-Bus

### 8.1 Einführung

Das I<sup>2</sup>C-Busprotokoll<sup>1</sup> wurde Anfang der 80er Jahre von der Firma Philips entwickelt und als serieller Bus für die Kommunikation zwischen Mikrocontrollern, ICs, sowie Ein- und Ausgabekomponenten konzipiert. Die steigende Verwendung von Mikrocontrollern in Consumer-Produkten erforderte damals eine neue Lösung, um die bislang verwendeten 8 oder 16 Bit breiten Datenbusse zu ersetzen. Durch den Einsatz des I<sup>2</sup>C-Standards konnten günstigere Controller mit weniger IOs verwendet werden. Dies machte platzsparende Platinen-Layouts möglich und führte damit zu einer Reduzierung der Produktionskosten. Mittlerweile wurde I<sup>2</sup>C nicht nur für Consumer-Produkte zu einem Standard, sondern auch für Steuerungs- und Diagnoseaufgaben in eingebetteten Systemen im industriellen Umfeld.



**Abb. 8.1.** Das I<sup>2</sup>C-Buskonzept.

I<sup>2</sup>C ist ein hierarchisches Bussystem, welches mit nur zwei Leitungen auskommt: dem Taktsignal SCL (*Serial Clock*) und einer weiteren Leitung zur Übertragung der Nutzdaten SDA (*Serial Data*) (vgl. Abbildung 8.1). Ein Ziel

<sup>1</sup> Engl. *Inter-Integrated Circuit*.

der Entwicklung war es, Komponenten verschiedenster Art mit geringem Aufwand an einen Bus anzubinden. Dies setzt die Verwendung eines einfachen Protokolls voraus, sodass keine separaten Treiberbausteine notwendig sind, um eine I<sup>2</sup>C-Schnittstelle zu realisieren. Dank dieser Eigenschaft lässt sich ein I<sup>2</sup>C-Treiber auch auf langsamen Mikrocontrollern implementieren, ohne dass zuviel Rechenkapazität für die Kommunikation aufgewendet werden muss. I<sup>2</sup>C ist ein bidirektionaler Zweidrahtbus mit Master-Slave-Konzept und Software-Adressierung. Die Datenübertragung erfolgt bitseriell und synchron. Jedes Datenbit auf der Datenleitung SDL wird mit einem Bit auf der Taktleitung SCL synchronisiert. Dadurch müssen die Taktzeiten nicht konstant sein, und es können sowohl langsamere als auch schnellere Busteilnehmer am gleichen Bus kommunizieren. Der Bus-Master koordiniert das Zeitverhalten und muss seinen Takt nach dem langsamsten Teilnehmer richten (vgl. Abschnitt 8.2).

In der ursprünglichen Spezifikation war eine Datenübertragung mit bis zu 100 kbit/s (*Standard Mode*) möglich. Durch eine Anpassung an die wachsenden Anforderungen wurde dieser Wert im Jahre 1992 auf 400 kbit/s (*Fast Mode*) erhöht. Diese beiden Modi fanden rasch Verbreitung und werden mittlerweile von vielen Geräten unterstützt. Im Gegensatz dazu fand die 1998 definierte Erweiterung auf 3,4 Mbit/s kaum Interesse, und es existieren nur wenige Bausteine, die diese Betriebsart verwenden.

Die Buslänge wird durch die maximal erlaubte Buskapazität von 400 pF auf nur zwei bis drei Meter begrenzt. Durch Verringerung der Kapazität, durch einen niedrigeren Bustakt oder durch Verwendung anderer physikalischer Übertragungsmedien ist es aber möglich, diese Grenze zu verschieben und damit Buslängen von bis zu 100 m und mehr zu erreichen. In Abschnitt 8.2.6 werden verschiedene Möglichkeiten der Buserweiterung beschrieben.

Neben der ursprünglichen Verwendung in HiFi-Consumer-Geräten zum Anschluss von Knöpfen, Wahlschaltern, Displays oder Echtzeituhren wird I<sup>2</sup>C auch in fast jedem PC verwendet, um die auf dem Mainboard verteilten Temperatursensoren einzulesen. Auch EEPROM-Chips auf EC- oder Krankenkassen-Karten sowie Sim-Karten-Chips sind Speicherbausteine, die als I<sup>2</sup>C-Slaves mit einem Mikrocontroller kommunizieren. Die im Vergleich zum CAN-Bus<sup>2</sup> relativ günstigen Schnittstellenbausteine ermöglichen die Verwendung eines I<sup>2</sup>C-Bussystemes auch in der Unterhaltungs- und Spielwarenelektronik. So werden z. B. in der neuesten Lego-Mindstorms-Generation Motoren und Sensoren auf Basis von I<sup>2</sup>C angesteuert.

Industrielle Feldbusse wie CAN, Profibus oder Interbus bieten andere Vorzüge wie höhere Übertragungsraten, Echtzeitfähigkeit, Priorisierung und Störunempfindlichkeit. Diese Eigenschaften werden tatsächlich aber auch in der Industrie oftmals nicht benötigt, sodass auch dort häufig die günstigere

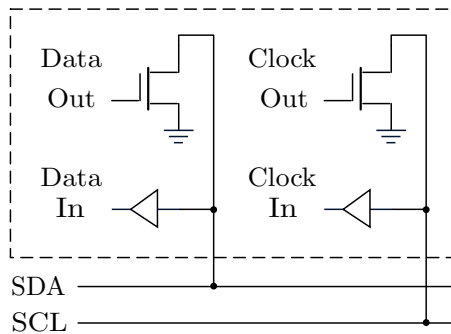
---

<sup>2</sup> Engl. *Controller Area Network*, vgl. Abschnitt 8.4.1.

I<sup>2</sup>C-Alternative zu finden ist. Ein Vergleich der verschiedenen seriellen Buskonzepte wird in Abschnitt 8.4 vorgestellt.

Der I<sup>2</sup>C-Bus weist folgende elektrische Eigenschaften auf:

- Geschwindigkeit 100 kbit/s im *Standard Mode*, 400 kbit/s im *Fast Mode* (dies entspricht einer SCL-Taktrate von 100 kHz bzw. 400 kHz)
- Maximale Buskapazität 400 pF
- SCL und SDA bidirektional, über Pull-Up Widerstände auf logisch „1“
- Buslänge regulär 2–3 m, je nach Erweiterung auch mehr als 100 m
- Multi-Master-fähig



**Abb. 8.2.** Eingangs- und Ausgangsschaltung für eine I<sup>2</sup>C-Anbindung als *Wired-AND*-Schaltung.

Die offizielle I<sup>2</sup>C-Spezifikation ist unter [I<sup>2</sup>C-Spezifikation 08] zu finden. Beide Leitungen SCL und SDA sind über Pull-Up-Widerstände mit der Betriebsspannung verbunden und befinden sich deshalb im Ruhezustand auf logisch „1“. Die als Open Collector ausgeführte Ausgangsstufe sperrt in diesem Zustand. Öffnet der Transistor, so wird die entsprechende Leitung auf Masse und damit auf logisch „0“ gelegt (vgl. Abbildung 8.2). Über die Eingangsschaltkreise werden anliegende Buspegel eingelesen und ausgewertet.

Abschnitt 8.2 erklärt das zugrunde liegende Prinzip der I<sup>2</sup>C-Datenübertragung und das Protokoll. Abschnitt 8.3 geht auf die Umsetzung einer I<sup>2</sup>C-Schnittstelle für verschiedene Systeme ein und erklärt die konkrete Anwendung. Abschnitt 8.4 stellt alternative Möglichkeiten der seriellen Kommunikation dem I<sup>2</sup>C-Bus gegenüber.



## 8.2 I<sup>2</sup>C-Datenübertragung

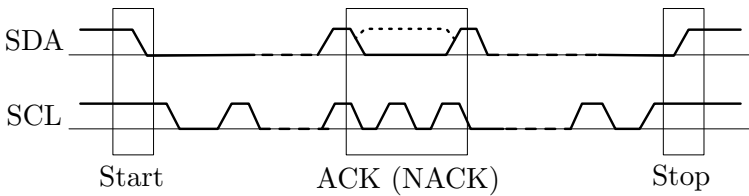
### 8.2.1 Konzept

Im Übertragungskonzept von I<sup>2</sup>C werden Busteilnehmer in Master und Slaves unterschieden. Ein Master initiiert eine I<sup>2</sup>C-Übertragung und ist für die Erzeugung des notwendigen Taktsignales verantwortlich. Er fordert durch entsprechende Adressierung eines Kommandos einen anderen Busteilnehmer auf, als Slave mit ihm zu kommunizieren, überwacht den Ablauf der Kommunikation und beendet diese. Grundsätzlich ist es möglich, mehrere Master an einem Bus zu betreiben. I<sup>2</sup>C kann deshalb auch als Multi-Master-Bus bezeichnet werden. Es ist allerdings dafür zu sorgen, dass stets nur ein Master zu jedem Zeitpunkt aktiv ist – eine Forderung, die durch bestimmte Kontrollmechanismen erfüllt werden kann. Besitzt ein Master Kontrollmechanismen wie Arbitrierung und Kollisionsvermeidung, so spricht man auch von einem Multi-Master. Beim Betrieb mit nur einem Master reicht die einfache Ausführung des Masters als Single-Master aus. In Abschnitt 8.2.4 werden weitere Einzelheiten zum Multi-Master-Konzept erläutert.

### 8.2.2 Steuersignale

Für die Koordination des Datentransfers stehen bei I<sup>2</sup>C sog. Steuersignale zur Verfügung. Durch eine fallende Flanke auf SDA während SCL=1 wird der Beginn einer Sequenz dargestellt. Durch diese Vorgabe wird der Bus vom Master als belegt gekennzeichnet, sodass sich kein anderer Baustein als Master anmelden kann. Eine steigende Flanke auf SDA während SCL=1 beendet eine Übertragung und gibt den Bus frei. Abbildung 8.3 veranschaulicht die Steuersignale. Wichtig ist dabei, dass während des Schreibens einer Start- oder Stop-Bedingung SCL dauerhaft auf „1“ liegt, um die Steuersignale korrekt interpretieren zu können. Unter Umständen müssen für einen Transfer zunächst Daten in einer Nachricht gesendet und anschließend in einer zweiten Nachricht gelesen werden, z. B. um ein bestimmtes Register auszulesen. Eine ungünstige Vorgehensweise wäre es, den Bus zwischen den Transfers durch das Senden einer Stop-Bedingung freizugeben, da dann ein anderer Multi-Master die Zuteilung erlangen und die Kommunikation unterbrechen könnte. Für diesen Fall steht die Möglichkeit zur Verfügung, ohne vorherigen Stop eine erneute Startbedingung zu senden (*Repeated Start Condition, RSC*). Auch wenn mehrere RSCs gesendet wurden, so wird der Bus durch Senden nur einer einzigen Stop-Bedingung wieder freigegeben.

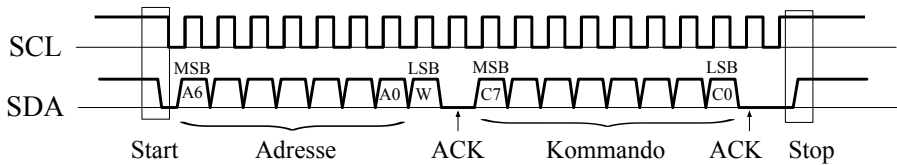
Daten werden über den I<sup>2</sup>C-Bus grundsätzlich als Einheiten von acht Bits übertragen. Die Maximalzahl von Bytes je Nachricht ist theoretisch nicht vorgegeben, in der Praxis aber durch das Protokoll der verwendeten Bausteine



**Abb. 8.3.** Steuersignale für Start- und Stop-Bedingungen sowie Bestätigungs-Bit.

begrenzt. Zuerst wird das höchstwertigste Bit (MSB) übermittelt, am Ende folgt das niederwertigste Bit (LSB).

Wurde nach dem Senden der Startbedingung das erste Byte vom Master übermittelt, so quittiert der Slave den Erhalt durch ein Bestätigungs-Bit (*Acknowledge*). Hierbei handelt es sich eigentlich um ein Steuersignal. Das Acknowledge-Bit wird jedoch wie ein Datensignal über SDL als logisch „0“ übertragen und liegt direkt als neuntes Bit nach acht übertragenen Datenbits an. Bleibt die Bestätigung aus (SDA=1), so signalisiert der Slave damit, dass er an der Kommunikation nicht mehr teilnimmt und kein weiteres Byte empfangen kann. In diesem Fall ist es wichtig, dass der Master das Senden einstellt, da alle weiteren Daten verloren gehen würden. Im Gegenzug muss der Master, wenn er von einem Slave einliest, nach jedem empfangenen Byte ein *Acknowledge* senden, um ein weiteres Byte anzufordern. Abbildung 8.4 veranschaulicht den Ablauf für das Senden zweier Daten-Bytes.



**Abb. 8.4.** Übertragung zweier Daten-Bytes: Slave-Adresse mit Angabe der Zugriffsart im niederwertigsten Bit, sowie ein Kommando-Byte.

### 8.2.3 Clock Stretching

Als *Clock Stretching* wird das Verfahren bezeichnet, das Low-Signal der Taktleitung künstlich zu verlängern. Diese Möglichkeit wird von Slaves genutzt, um einem Master mitzuteilen, dass die Daten nicht mit der aktuellen Frequenz verarbeitet werden können. Dadurch wird ein Datenverlust verhindert und nachfolgende Daten können korrekt gelesen werden. Dies setzt allerdings eine Überwachung der SCL-Leitung durch den Master voraus, um auf ein *Clock Stretching* angemessen reagieren zu können.

### 8.2.4 Multi-Master-Betrieb

In der einfachen Variante mit nur einem Master am Bus ist eine Zuteilung oder Arbitrierung nicht notwendig. Solange kein Clock Stretching stattfindet, wäre es auch möglich, auf das Einlesen von SCL komplett zu verzichten und lediglich die Befehle auf den Bus zu schreiben. Im Multi-Master-Betrieb jedoch müssen alle Master am Bus zusammenarbeiten und eine Zuteilung für einen jeweils aktiven Master aushandeln. Während ein Master kommuniziert, müssen die anderen Master warten, bis der Bus freigegeben wird. Sollten zwei Master genau gleichzeitig zu senden beginnen, so muss dieser Fall erkannt und gelöst werden. Folgende Fähigkeiten unterscheiden deshalb den Multi-Master vom Single-Master:

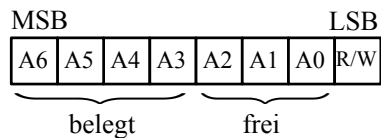
- **Belegtheitserkennung:** Bevor ein Master zu senden beginnt, ist zu überprüfen, ob während einer bestimmten Zeit (max. eine Periode) das SCL-Signal den Wert „0“ trägt. Ist dies der Fall, so findet momentan eine Kommunikation statt. Dann muss die nächste Stop-Bedingung abgewartet werden, bevor erneut gesendet werden darf.
- **Kollisionserkennung:** Beginnen zwei Master gleichzeitig mit der Kommunikation, so bekommt das langsamere Gerät den Bus zugeteilt. Dabei hat jeder Master zu überprüfen, ob die von ihm erzeugte „1“ auf der SCL-Leitung tatsächlich anliegt. Ist dies nicht der Fall, so verlängert ein anderer Master den Takt und die anderen Master müssen sich zurückziehen. Wie bereits beim *Clock Stretching* so setzt sich auch hier die langsamere Periode durch.

Beim Betrieb mehrerer Master ist es unbedingt notwendig, dass alle Master im Multi-Master-Betrieb arbeiten und die genannte Funktionalität besitzen.

### 8.2.5 Adressierung

Die Baustein-Adressierung erfolgt in der Regel mit sieben Bits, welche der Master zusammen mit der Lese- oder Schreibinformation als erstes Byte einer Nachricht sendet (siehe Abbildungen 8.4 und 8.5). Laut I<sup>2</sup>C-Spezifikation ist ebenfalls eine Adressierung mit zehn Bits vorgesehen, um bis zu 1 024 Bausteine an einem Bus betreiben zu können. Tatsächlich ist dies in der Praxis aber kaum relevant, da nur relativ wenige 10-Bit-Bausteine existieren.

Von den mit sieben Bits theoretisch möglichen 128 Adressen sind tatsächlich nur 112 verfügbar, da einige Adressen reserviert sind (vgl. Tabelle 8.1). Für die Bausteinadressierung stehen in der Regel die drei niederwertigen Bits der Adresse zur Verfügung, welche oftmals auf Pins herausgeführt sind und über Jumper gesetzt werden können. Die oberen vier Bits der Bausteinadresse sind



**Abb. 8.5.** Erstes Byte einer I<sup>2</sup>C-Nachricht bestehend aus 7-Bit-Adresse und Lese- oder Schreibanforderung.

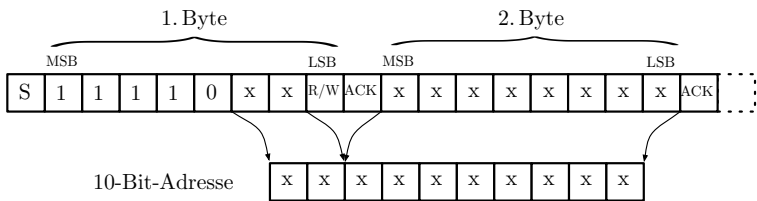
meist durch den Hersteller fest vorgegeben und definieren die jeweilige Bausteinfamilie. Für manche Bausteine existieren verschiedene Ausführungen dieser 4-Bit-Codierung, um auch einen Betrieb von mehr als acht Bausteinen des gleichen Typs an einem Bus zu ermöglichen.

Adresse	R/W-Bit	Verwendung
0000000	0	General-Call-Adresse
0000000	1	Start-Byte
0000001	x	CBUS-Adressen
0000010	x	Reserviert zur Unterscheidung der Busformate
0000011	x	Reserviert für zukünftige Erweiterungen
00001xx	x	Umschaltung auf High-Speed-Modus
11110xx	x	10-Bit-Adressierung
11111xx	x	Reserviert für zukünftige Erweiterungen

**Tabelle 8.1.** Reservierte I<sup>2</sup>C-Adressen.

Sollten die Adressen nicht ausreichen, so kann der Bus mit Multiplexer-Bausteinen aufgesplittet werden. Nähere Informationen dazu werden in Abschnitt 8.2.6 beschrieben.

Über die 10-Bit-Adresse wird eine 10-Bit-Übertragung angekündigt, deren Slave-Adresse aus den zwei folgenden Bytes gemäß Abbildung 8.6 zusammengesetzt wird. Diese Reservierung hat den Vorteil, dass sowohl 7-Bit- als auch 10-Bit-Bausteine gleichzeitig an einem Bus betrieben werden können.



**Abb. 8.6.** Adressierung eines 10-Bit-Bausteins. Nach der Übertragung des ersten Teils der Adresse folgen unter Umständen mehrere Bestätigungen der Slave-Bausteine, nach dem zweiten Teil nur die des angesprochenen Bausteins.

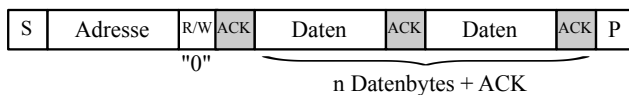
Wurde das Adress-Byte vom zugehörigen Slave korrekt empfangen, so folgt wie nach jedem empfangenen Byte eine Bestätigung. Grundsätzlich sind drei Möglichkeiten des Datentransfers zu unterscheiden – Abbildung 8.7 veranschaulicht die Varianten:

1. Der Master sendet Daten an einen Slave.
2. Der Master liest Daten von einem Slave.
3. Kombiniertes Schreib- Lesezugriff: Der Master sendet Daten und liest dann umgehend die Antwortdaten.

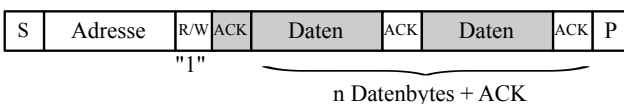
Erfolgt ein Schreibvorgang ( $R/W=0$ ), so sendet der Master nach der Übertragung der Adresse weiterhin Daten, bis er den Vorgang mit einer Stop-Bedingung abschließt. Bleibt die Bestätigung aus, so wird die Übertragung vom Master durch eine Stop-Bedingung abgebrochen, um den Bus wieder freizugeben. Sollen Daten empfangen werden ( $R/W=1$ ), so kehrt sich nach dem Schreiben des Adress-Bytes der Datenfluss um. Der Master bestätigt nun seinerseits durch ein *Acknowledge* den korrekten Empfang der Daten-Bytes. Die Übertragung wird wieder vom Master beendet (also nicht, wie man vielleicht vermuten könnte, vom Slave).

In einer dritten Variante wird nach dem Schreiben eines oder mehrerer Daten-Bytes nahtlos über eine *Repeated-Start-Condition* ein anschließender Lesevorgang begonnen. Wie zu Beginn in Abschnitt 8.2 erklärt, wird der Bus dadurch nicht freigegeben und es erfolgt eine kombinierte Schreib-Lese-Operation. Diese Variante wird beispielsweise benötigt, wenn in Slave-Bausteinen ein bestimmtes Register ausgewählt und sofort gelesen werden soll, ohne den Bus zwischen den Transaktionen freizugeben.

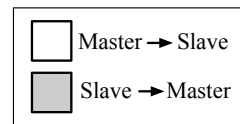
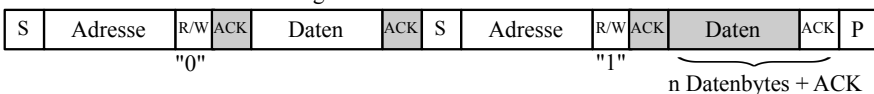
1. Master sendet Daten an Slave



2. Master liest Daten von Slave



3. Kombiniertes Schreib- Lesezugriff

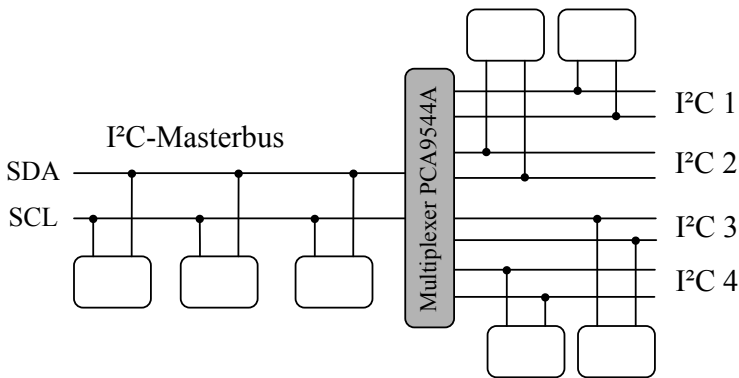


**Abb. 8.7.** Verschiedene Varianten des Datentransfers: Schreiboperation, Leseoperation und kombinierter Datentransfer.

### 8.2.6 I<sup>2</sup>C-Buserweiterungen

Eine Erweiterung des I<sup>2</sup>C-Busses kann hinsichtlich der maximal erlaubten Anzahl von Bausteinen gleichen Typs oder der maximalen Übertragungslänge erfolgen. Im ersten Fall werden sog. I<sup>2</sup>C-Multiplexer verwendet, um weitere Unterbusse mit eigenem Adressraum zu erzeugen. Im zweiten Fall wird mithilfe sog. Bus-Extender-Bausteine die maximale Distanz zwischen den I<sup>2</sup>C-Komponenten auf mehrere hundert Meter erhöht.

#### Aufteilung in mehrere I<sup>2</sup>C-Unterbusse



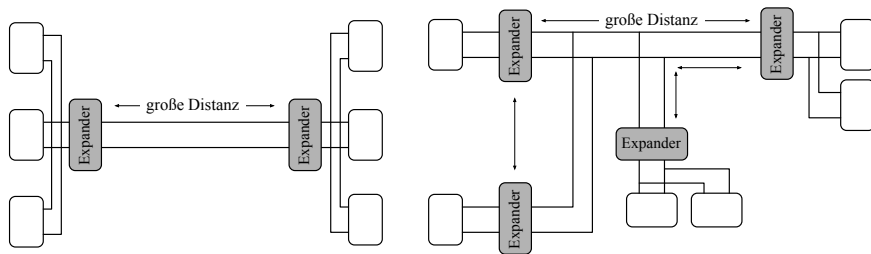
**Abb. 8.8.** Erweiterung der I<sup>2</sup>C-Bus-Topologie mit einem Multiplexer des Typs PCA9544A. Durch bis zu vier separate Unterbusse können Bausteine mit der gleichen Adresse mehrmals im System verwendet werden.

Falls es bei einer gegebenen Beschränkung auf nur drei frei wählbare Adressbits notwendig wird, mehr als acht Bausteine eines Typs zu betreiben, so kann der I<sup>2</sup>C-Bus mithilfe eines Multiplexers in mehrere Unterbusse aufgeteilt werden. Der Philips PCA9544A bietet bis zu vier I<sup>2</sup>C-Kanäle, wovon jeweils einer mit dem Hauptbus verbunden sein kann (vgl. Abbildung 8.8). Die Auswahl des aktuellen Kanals geschieht über ein Kontrollregister. Adressierung und Zugriff erfolgen wie bei einem herkömmlichen I<sup>2</sup>C-Slave-Baustein. Anzumerken ist, dass die Unterteilung zu Lasten der Übertragungsgeschwindigkeit in den Einzelbussen geht. Unter Umständen kann es sinnvoll sein, die Geräte anhand ihrer Lese- und Schreibraten auf Unterbusse zu gruppieren, um ein häufiges Umschalten zu vermeiden. Abschnitt 9.8 zeigt die Programmierung und Verwendung eines PCA9548A mit acht Kanälen.

## Verlängerung der Reichweite

I<sup>2</sup>C ist laut Spezifikation auf eine maximal zulässige Buskapazität von 400 pF beschränkt (Leitung + Eingangskapazitäten der Teilnehmer). Diese Größe entspricht einer Buslänge von nur wenigen Metern. Der durch die Tiefpasseigenschaften entstehende, sägezahnartige Spannungsverlauf bewirkt, dass die Pegel nicht mehr zuverlässig erkannt werden. Eine Verlangsamung des Taktes und ein dadurch verlängertes Anliegen der Pegel kann die Zuverlässigkeit erhöhen. Einige Anwender berichten von einem I<sup>2</sup>C-Busbetrieb bis zu 100 m Länge, allerdings bei einem Takt von nur 500 Hz. Dies ist nicht immer ein gangbarer Weg, die Vorgehensweise kann aber zumindest angewandt werden, um die Übertragungssicherheit weiter zu erhöhen.

Eine elegantere Lösung zur Verlängerung der Reichweite ist die Verwendung von Treiber- bzw. Pufferbausteinen. In der folgenden Beschreibung werden diese unter dem Begriff *Expander* zusammengefasst. Damit ist eine Busverlängerung auf 200–250 m bei 100–400 kHz Taktrate möglich. Die Arbeitsweise basiert grundsätzlich auf einer Verringerung des Ausgangswiderstandes der Treiber oder der Verwendung einer anderen physikalischen Übertragungsebene. Die Auswahl sollte abhängig von der vorhandenen Bustopologie und den zu überbrückenden Distanzen erfolgen (vgl. Abbildung 8.9).



**Abb. 8.9.** I<sup>2</sup>C-Bus-Topologien: Punkt-zu-Punkt-Verbindung zweier Gruppen mit großer Distanz (links) und Anordnung in Baumstruktur mit großem Abstand zwischen Gruppen und Einzelbausteinen (rechts). Der Einsatz von Treiber- oder Pufferbausteinen erfolgt an markanten Punkten des Netzes.

Folgende Varianten sind denkbar um die Reichweite zu erhöhen:<sup>3</sup>

- Aktiver Pull-Up-Widerstand LTC1694-1:  
Dieser aktive Pull-Up-Widerstand verringert den Einfluss der Leitungskapazität, indem bei positivem Pegel mit einem Strom von bis zu 2,2 mA unterstützt wird. Bei negativem Pegel erfolgt kein Eingriff. Die Aktivierung dieser Unterstützung findet durch einen externen Pull-Up-Widerstand

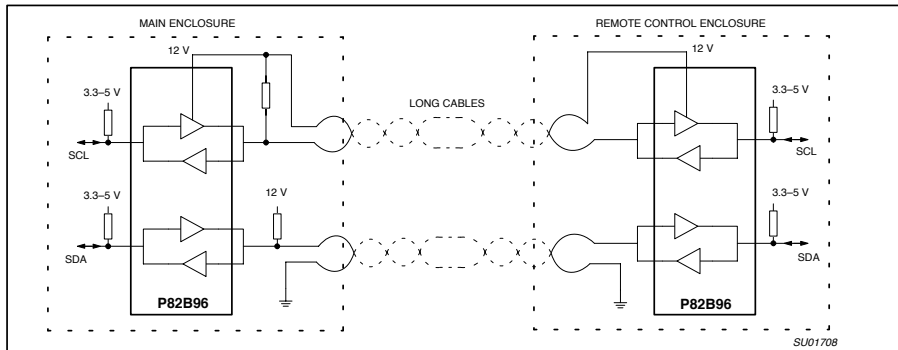
<sup>3</sup> Die Datenblätter der genannten Bausteine befinden sich im Verzeichnis `<embedded-linux-dir>/datasheets/`.

statt, welcher gemäß den Vorgaben im Datenblatt des LTC1694-1 auszulegen ist. Je nach verwendeter Struktur sind aktive Pull-Ups am Anfang und am Ende einer langen Verbindung zwischen I<sup>2</sup>C-Gruppen (Punkt-zu-Punkt) oder an Verzweigungspunkten zu setzen (Baumstruktur). Als Faustregel kann gelten, dass die Entfernung zwischen I<sup>2</sup>C-Baustein und aktivem Pull-Up eine Länge von 20 m nicht überschreiten sollte. Dieser Baustein wird unter anderem im Leitungstreiber-Modul von Conrad verwendet (vgl. Komponentenliste in Tabelle E.2). Um die Pull-Up-Leistung über 2,2 mA hinaus weiter zu vergrößern, können mehrere LTC1694-1 parallel geschaltet werden. Im Vergleich zu den anderen vorgestellten Bausteinen findet keine Pegelkonvertierung statt. Physikalisch folgt das Busnetz weiterhin auch hinsichtlich der Pegel dem I<sup>2</sup>C-Standard. Die Taktrate des I<sup>2</sup>C-Busses ist bei Verwendung dieses Bausteins allerdings auf 100 kHz beschränkt.

- **Bus-Extender Philips P82B715:**  
Der Bus-Extender arbeitet als Schnittstellenbaustein zwischen einem regulären I<sup>2</sup>C-Bus und einem gepufferten Bus mit niedriger Impedanz und bewirkt eine Reduzierung des Einflusses der Kabelkapazitäten auf 1/10. Für das Gesamtnetz sind deshalb höhere Systemkapazitäten von bis zu 3 000 pF zulässig, wodurch bei einer hohen Übertragungsrate von 400 kHz Buslängen bis 50 m realisiert werden können. An einem gepufferten Bus ( $S_x, S_y$ ) ist kein zuverlässiger Betrieb regulärer I<sup>2</sup>C-Bausteine möglich. Entsprechend muss allen angeschlossenen Geräten oder Gruppen zur Kommunikation mit diesem Bussystem ein Bus-Extender vorgeschaltet werden. Laut Datenblatt sind für einen optimalen Betrieb Pull-Up-Widerstände für die jeweiligen I<sup>2</sup>C-Bussektionen und für das gemeinsame Subsystem auszulegen.
- **Dual-Bus-Puffer mit Philips P82B96:**  
Mit dem Treiberbaustein P82B96 von Philips ist eine sichere Übertragung der I<sup>2</sup>C-Daten über weite Strecken bis zu 250 m möglich. Die Takraten betragen hierbei bis zu 400 kHz. Die Pegel werden dabei je nach verwendeter Versorgung auf 12 V erhöht – eine Maßnahme, die die Übertragungssicherheit und -distanz im Vergleich zu den bisher vorgestellten Lösungen wesentlich verbessert. Aufgrund der Unterteilung in Rx- und Tx-Pfade können auch unidirektionale Komponenten wie bspw. Optokoppler zwischengeschaltet werden. Durch Zusammenschluss von Rx und Tx sowie Ry und Ty erhält man einen vollwertigen I<sup>2</sup>C-Bus als Subsystem, welcher wiederum verzweigt werden kann (vgl. Abbildung 8.10). Aufgrund der höheren Pegel im Subsystem sind auch hier P82B96-Treiberbausteine vor jedem I<sup>2</sup>C-Gerät bzw. jeder Gruppe notwendig. Aufgrund der genannten Vorteile ist für die Auslegung neuer Schaltungen dieser Baustein dem P82B715 vorzuziehen.



Neben den I<sup>2</sup>C-Signalleitungen sind in der Regel auch Versorgungsspannung und Masse mitzuführen. Eine Verwendung paarweise verdrehter Kabel<sup>4</sup> liefert beste Ergebnisse um Störungen bspw. von magnetischen Wechselfeldern auszugleichen. Der Hersteller Philips schlägt diese Kabelbelegung in der *Application Note AN255-02*<sup>5</sup> vor. In diesem Informationsblatt werden zudem I<sup>2</sup>C-Komponenten wie Repeater, Hubs und Expander verglichen und weiterführende Hinweise zur richtigen Auswahl gegeben.



**Abb. 8.10.** Treiberbaustein Philips P82B96: Vorschlag für die Belegung von Twisted-Pair-Kabeln zur Überbrückung von Strecken bis zu 250 m. Quelle: Philips Application Note AN255-02.

### 8.3 I<sup>2</sup>C-Anbindung

Aufgrund der Einfachheit des I<sup>2</sup>C-Protokolls und der geringen zeitlichen Anforderungen kann ein I<sup>2</sup>C-Treiber vergleichsweise einfach komplett in Software realisiert werden. Hierzu sind lediglich zwei I/O-Leitungen notwendig, um Taktsignal (SCL) und Datensignal (SDA) zu erzeugen. Diese I/O-Leitungen können direkt am Prozessor in Form von frei verfügbaren *General-Purpose-IOs* (GPIOs) entnommen werden – allerdings ist dafür eine Kontaktierung der oftmals schwer zugänglichen Prozessorpins notwendig. Falls der Prozessor in Ball-Grid-Technik aufgelötet wurde, so sind die Pins generell nicht direkt zu erreichen. Bei einer Verwendung der I/O-Leitungen des Prozessors ist zu

<sup>4</sup> Es wird jeweils eine I<sup>2</sup>C-Leitung mit einer Versorgungsleitung verdreht. Kabel dieser Art werden im Netzbereich verwendet und als sog. Twisted-Pair-Kabel vertrieben.

<sup>5</sup> Verfügbar unter `<embedded-linux-dir>/datasheets/` oder auf der Seite des Herstellers [http://www.nxp.com/acrobat\\_download/applicationnotes/AN255\\_2.pdf](http://www.nxp.com/acrobat_download/applicationnotes/AN255_2.pdf).

berücksichtigen, dass Takt- und Datenleitung bei logischer „1“ die Versorgungsspannung des Bausteins führen. Bei modernen Prozessoren beträgt diese oft nur 3,3 V. Da die Pins normalerweise tolerant gegenüber angelegten 5 V sind, ist ein gemischter Betrieb möglich. Zu beachten ist allerdings, dass die Leitungslänge durch den geringeren Pegel eingeschränkt wird. Ein Pegelwandler wie der Philips PCA9515 (I<sup>2</sup>C-Puffer mit Pegelwandlung) isoliert Busse mit verschiedenen Spannungen und kann in diesem Fall Abhilfe schaffen.

Auch über die Steuerleitungen der seriellen Schnittstelle kann mit wenig Aufwand ein I<sup>2</sup>C-Bus realisiert werden. Hierzu ist lediglich eine Pegelanpassung notwendig, um die 12 V der seriellen Schnittstelle auf 5 V abzubilden. Bezugsquellen für diese Adapter sind in Tabelle E.2 aufgelistet.<sup>6</sup> Die Voraussetzung für den Einsatz dieser Technik ist allerdings die Fähigkeit zum Hardware-Handshaking bzw. das Vorhandensein der entsprechenden Steuerleitungen. Diese Leitungen sind zwar bei den meisten PC-basierten Systemen verfügbar, fehlen aber häufig bei Embedded-Boards.

Bei der NSLU2 von Linksys oder dem VisionSystems Alekto (vgl. Kapitel 4 bzw. 5) ist auf der Platine jeweils eine I<sup>2</sup>C-Echtzeituhr vorhanden, welche im ausgeschalteten Zustand über eine Pufferbatterie versorgt wird und Datum sowie Uhrzeit weiterzählt. Über GPIO-Pins der Prozessoren wird ein 100 kHz I<sup>2</sup>C-Master emuliert, um mit der Echtzeituhr zu kommunizieren. I<sup>2</sup>C-Treiber sind bei diesen Debian-Systemen bereits enthalten<sup>7</sup>, sodass ein Zugriff auf die am I<sup>2</sup>C-Bus angeschlossenen Bausteine direkt möglich ist. In Kapitel 2.2 wird im Rahmen der Hardware-Beschreibung auch das Herausführen des NSLU2-I<sup>2</sup>C-Anschlusses erklärt; beim Alekto-Embedded-PC liegt der I<sup>2</sup>C-Bus bereits auf einem nach außen geführten Steckverbinder. Abschnitt 8.3.2 beschreibt die Verwendung des bereits vorhandenen I<sup>2</sup>C-Busses für NSLU2 und Alekto.

Über die sog. IO-Warrior-Bausteine der Fa. Code Mercenaries [Codemercs 08] lassen sich auch über die USB-Schnittstelle relativ einfach I/O-Erweiterungen, I<sup>2</sup>C und SPI realisieren (vgl. Abschnitt 7.5). Diese Möglichkeit ist insbesondere für Geräte interessant, die nicht wie NSLU2 oder Alekto von Haus aus über einen I<sup>2</sup>C-Bus verfügen. Basierend auf dem HID-Standard für *Human Interface Devices* ist die Anbindung der IO-Warrior besonders einfach. Ein offenes Treibermodul wird mitgeliefert und kann für Debian nach Anleitung installiert werden (vgl. Anhang 7.5.2). Für andere Linux-Distributionen ist etwas mehr Aufwand notwendig – die Vorgehensweise für OpenWrt ist in Abschnitt 3.6 beschrieben. Generell steht mit dem IO-Warrior eine einheitliche Lösung zur Verfügung, um Geräte mit USB-Schnittstelle auch mit einer I<sup>2</sup>C-Busschnittstelle zu versehen.

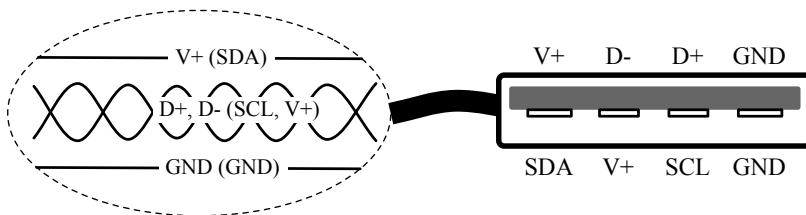
<sup>6</sup> Unter [http://www.robotikhardware.de/download/rn\\_pc\\_i2c.pdf](http://www.robotikhardware.de/download/rn_pc_i2c.pdf) ist eine Beispielschaltung verfügbar.

<sup>7</sup> Die I<sup>2</sup>C-Treiber sind seit Linux-Version 2.6 im Kernel enthalten. Ein separates I<sup>2</sup>C-Paket wird nicht mehr benötigt.

### 8.3.1 I<sup>2</sup>C-Steckverbindung

Neben den Datenleitungen SCL und SDA ist es sinnvoll, in einer I<sup>2</sup>C-Steckverbindung zusätzlich eine 5 V-Versorgungsspannung und die Masse mitzuführen. Die Versorgung soll hierbei lediglich die logischen Bauteile und kleine Verbraucher am Bus speisen – 500 mA Maximalstrom sind hierfür ausreichend. Eine Schirmung und eine kleine Kapazität der Kabel ist wegen des genannten Einflusses auf die Übertragungslänge wünschenswert. Da USB-Verbindungen die geforderten Eigenschaften erfüllen und preisgünstig zu beziehen sind, kann dieser Hardware-Standard auch für I<sup>2</sup>C verwendet werden. Auch wenn I<sup>2</sup>C keine differentielle Übertragung verwendet, so ist eine Verdrillung dennoch gut geeignet, um Gleichtaktstörungen zu unterdrücken. Optimal wäre hierbei eine paarweise Verdrillung jeder I<sup>2</sup>C-Leitung, bei USB-Kabeln wird jedoch nur das Paar (D+/D-) verdreht. Das höherfrequente SCL-Signal soll den Vorzug erhalten und wird mit V+ verdreht, während SDA und Masse regulär übertragen werden.

Die Übertragung der Versorgungsspannung entsprechend der USB-Leitungsbelegung erscheint zunächst naheliegend, muss aber auf jeden Fall vermieden werden. SDA und SCL würden in diesem Fall auf D+ und D- verdreht übertragen und sich gegenseitig beeinflussen. Abbildung 8.11 zeigt die I<sup>2</sup>C-Belegung für USB-Kabel, wie sie im Anwendungshinweis AN255\_02 von Philips vorgeschlagen wird.



**Abb. 8.11.** USB-Steckverbindung für den I<sup>2</sup>C-Bus: Blick auf eine USB-Buchse vom Typ A mit Original-USB-Belegung und alternativer I<sup>2</sup>C-Zuordnung.

Der Vorteil bei der Verwendung von USB-Steckverbindungen für den I<sup>2</sup>C-Bus ist die Verfügbarkeit von verschiedenen Steckergrößen, die auf die jeweilige Anwendung abgestimmt werden können. Weiterhin können geschirmte und fertig konfektionierte Kabel einfach und preisgünstig bezogen werden [Reichelt 08]. Entsprechend wird diese Art der Steckverbindung auch für den Anschluss der in Kapitel 9 vorgestellten I<sup>2</sup>C-Geräte verwendet.

### 8.3.2 Verwendung des I<sup>2</sup>C-Busses bei NSLU2 und Alekto

Bei den beiden genannten Systemen wird die I<sup>2</sup>C-Schnittstelle üblicherweise als Gerätedatei `/dev/i2c-0` repräsentiert. Die notwendigen Treiber zur Umsetzung der Bitfolgen auf die physikalischen Leitungen sind bereits geladen, sodass ein Zugriff auf Dateisystemebene mit herkömmlichen Lese- und Schreiboperationen erfolgen kann.

Das mitgelieferte Beispiel `<embedded-linux-dir>/examples/iic/iic_native` zeigt den einfachen I<sup>2</sup>C-Zugriff mit C-Funktionen:

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <linux/i2c-dev.h>
#include <linux/i2c.h>

int main(){

    int fd,n;
    char buf[10];

    // use this address for NSLU2 RTC clock
    int addr = 0x6F;

    // use this address for Alekto RTC clock
    //int addr = 0x68;

    if ((fd = open("/dev/i2c-0", O_RDWR)) < 0) {
        printf("open error! %d\n",fd);
        return 1;
    }

    if (ioctl(fd, I2C_SLAVE, addr) < 0) {
        printf("address error!\n");
        return 1;
    }

    buf[0] = 0x00;

    printf("write: ");
    n = write(fd, buf, 1);

    if (n != 1)
        printf("error! %d\n",n);
    else
        printf("0x%x, 0x%x\n", buf[0], buf[1]);

    printf("read: ");
    n=read(fd, buf, 1);

    if (n != 1)
        printf("read error! %d\n", n);
    else
        printf("0x%x\n", buf[0]);

    close(fd);
    return 0;
}
```

Außer der Header-Datei für die Standard-Dateioperationen sind für den Zugriff auf den I<sup>2</sup>C-Bus die Header-Dateien `linux/i2c-dev.h` und `linux/i2c.h`

einzubinden. Nach dem Öffnen des Gerätes mit `open()` wird im Erfolgsfall (Rückgabewert größer 0) ein Filedescriptor zurückgegeben, der das Gerät referenziert. In diesem Fall wird der Bus mit der Option `O_RDWR` für Lese- und Schreibzugriffe geöffnet.

Wie in den Grundlagen zur I<sup>2</sup>C-Kommunikation in Abschnitt 8.1 beschrieben, muss zu Beginn jeder Byte-Folge die Adresse des Bausteins gesendet werden. Da diese Aufgabe zukünftig der Treiber übernimmt, und der Anwender lediglich die nachfolgenden Daten-Bytes sendet bzw. liest, muss dem Treiber die Bausteinadresse bekannt sein. Das Setzen der aktuellen Slave-Adresse geschieht mit dem `ioctl()`-Befehl.

Diese Zuordnung bleibt aktiv bis eine neue Adresse übermittelt wird. Ein Rückgabewert = 0 bestätigt den Erfolg der Aktion. Wird anstatt der Option `I2C_SLAVE` der Wert `I2C_SLAVE_FORCE` verwendet, so fordert die Anwendung Zugriff auf einen I<sup>2</sup>C-Baustein, auch wenn der Kernaltreiber diesen gerade in Bearbeitung hat. Diese Option ist mit Vorsicht anzuwenden, hilft allerdings beim Testen ab und an, um nach einem Programmabsturz wieder Zugriff auf die I<sup>2</sup>C-Bausteine zu erlangen. Nun können mit den Befehlen `read()` und `write()` Daten-Bytes geschrieben bzw. gelesen werden, wobei die I<sup>2</sup>C-Adresse nicht mehr angegeben werden muss. Als Attribute werden Filedescriptor `fd`, ein Zeiger `buf` vom Typ `char*` und die Anzahl Bytes übergeben.

I<sup>2</sup>C definiert keine maximale Länge der Nachrichten. Je nach Komplexität der Befehle reichen aber oft acht Daten-Bytes für das Senden und Empfangen aus. Als Rückgabewert dient die Anzahl der verschickten Bytes. Das Beispiel kann an der Kommandozeile mit folgenden Befehlen übersetzt und gestartet werden:

```
$ make
$ sudo ./iic_native
```

Üblicherweise ist der Zugriff auf `/dev/i2c-0` standardmäßig nur dem Benutzer `root` gestattet. Wenn der aktuelle Benutzer in der `sudoer`-Liste eingetragen ist, so kann diese Hürde mit `sudo` umgangen werden (zur Freischaltung dieser Möglichkeit vgl. auch Abschnitt A.4.5). Alternativ können aber auch die Zugriffsrechte auf das Gerät angepasst werden (vgl. Abschnitt A.4.4).

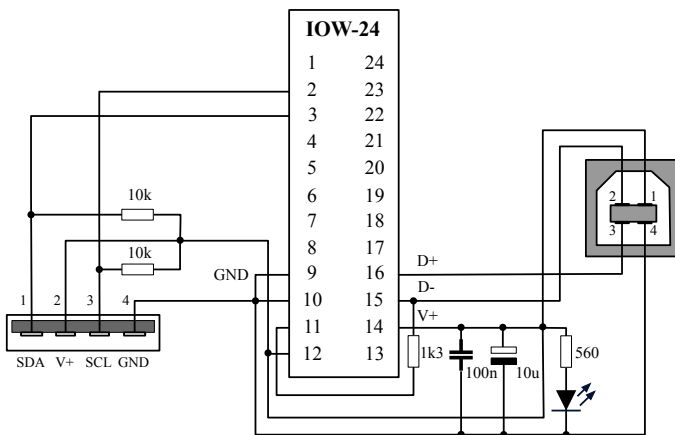
Bei der Ausführung des Beispielprogrammes stellt man fest, dass der Befehl `write()` scheinbar auch ohne angeschlossene Geräte ein Byte absetzen kann. Dies ist nur möglich, wenn ein Partner vorhanden ist, der die empfangenen Bytes durch ein Acknowledge bestätigt. Weiterhin kann als Antwort darauf mit dem Befehl `read()` ein Daten-Byte empfangen werden. Tatsächlich ist bereits ein Gerät mit Adresse `0x6F` am Bus der NSLU2 vorhanden. Es handelt sich um eine Echtzeituhr vom Typ Intersil X1205.<sup>8</sup> Dieser batteriegepufferte I<sup>2</sup>C-Baustein errechnet fortlaufend Datum und Uhrzeit und kann über I<sup>2</sup>C abgefragt oder neu gesetzt werden.

<sup>8</sup> Beim Alekto ist dies eine Echtzeituhr vom Typ Dallas DS1337 mit Adresse `0x68`. Die Adresse muss im Quelltext umgestellt werden.

Am Ende eines Programmes sollte der angeforderte Dateizugriff auf das Gerät mit `close()` wieder gelöst werden. Im Kapitel *Inter-IC-Buskomponenten* wird der Zugriff auf weitere Komponenten ausgedehnt, und es wird eine C++-Bibliothek für verschiedene I<sup>2</sup>C-Bausteine entwickelt. Die Grundlage dafür bilden die im vorliegenden Abschnitt vorgestellten Mechanismen.

### 8.3.3 I<sup>2</sup>C-Busanbindung über einen IO-Warrior-Baustein

Die IO-Warrior-Module der Firma Code Mercenaries wurden bereits in Abschnitt 7.5 vorgestellt [Codemerces 08]. Im Folgenden wird davon ausgegangen, dass ein IO-Warrior-Modul IOW24, IOW40 oder IOW56 am System angeschlossen ist, da nur diese eine I<sup>2</sup>C-Schnittstelle besitzen. Weiterhin müssen die Treiber installiert und Geräte in der Form `/dev/iowarrior<xy>` angelegt sein.<sup>9</sup> Neben der Verwendung der Starterkit-Platinen können die IO-Warrior-ICs auch separat bestellt werden; sie müssen zum Betrieb lediglich mit Pufferkondensator und Steckverbindungen versehen werden. Abbildung 8.12 zeigt die entsprechende Beschaltung.



**Abb. 8.12.** Schaltplan zum Anschluss eines IOW24-Bausteines.

Im folgenden Abschnitt wird lediglich auf die I<sup>2</sup>C-Ansteuerung des Bausteins genauer eingegangen. Die anderen Funktionen wie SPI-Kommunikation, LCD-Unterstützung und Verwendung der IOs können im Datenblatt unter `<embedded-linux-dir>/datasheets/iowarrior24_40.pdf` nachgeschlagen werden.

<sup>9</sup> Bei manchen Systemen werden die IO-Warrior-Geräte dateien auch unter `/dev/usb/iowarrior<xy>` angelegt.

Die Verwendung der mitgelieferten IO-Warrior-Bibliothek `libiowkit.so` stellt für IO-Warrior-Bausteine komfortable Funktionen für die Schnittstellenzugriffe zur Verfügung. Die Unterstützung der I<sup>2</sup>C-Funktionalität ist allerdings nur unzureichend realisiert, und entsprechend werden die dynamischen Bibliotheken in den vorliegenden Beispielen nicht verwendet. Stattdessen erfolgt der Zugriff direkt über das Kernelmodul und dessen Basisfunktionen (vgl. hierzu auch die Datei `IOWarriorHowTo.pdf` im jeweiligen Modulverzeichnis).

### 8.3.4 Die IO-Warrior-I<sup>2</sup>C-Bibliothek

Die Erweiterungen in `iowarrior_i2c.c` und `iowarrior_i2c.h`<sup>10</sup> ergänzen die fehlenden Funktionen und stellen ein einfaches I<sup>2</sup>C-Interface zur Verfügung. Die API orientiert sich grob an der Schnittstelle, die in Abschnitt 8.3.2 für die direkte Ansteuerung des I<sup>2</sup>C-Busses für die NSLU2 bzw. den Alekto vorgestellt wurde. Die Implementierung ist bewusst in C erfolgt, um eine Einbindung in C-basierte Anwendungen zu ermöglichen. Folgende Funktionen sind enthalten:

```
void iow_print_warriors(const char* basename);
```

Diese Funktion listet alle IO-Warrior-Bausteine auf, die im System unter `basename` vorhanden sind (z. B. `/dev/iowarrior`). Hiermit können Informationen wie bspw. die Seriennummer der angeschlossenen IO-Warriors angegeben werden.

```
int iow_find_special_descriptor(const char *serial, const char*
    basename);
```

Diese Funktion gibt den Filedescriptor für den Zugriff auf den Spezialmodus zurück, über den auch I<sup>2</sup>C angesprochen wird. Der IO-Warrior-Baustein ist über Seriennummer und Basisnamen anzugeben, und entsprechend lassen sich so auch mehrere Platinen unterscheiden. Wenn kein Baustein mit der angegebenen Seriennummer gefunden wird, so wird `-1` zurückgeliefert.

```
int iow_i2c_write(char address, char *buf, int size, int
    report_size, int fd);
```

Die Funktion schreibt an die I<sup>2</sup>C-Adresse `address` eine Anzahl von `size` Daten-Bytes, beginnend an Stelle `buf`. Die Länge eines Reports muss zuvor mit `iow_i2c_init()` ermittelt werden, der Filedescriptor mit `iow_find_descriptor()`. Als Rückgabewert wird im Erfolgsfall die Anzahl geschriebener Bytes geliefert, sonst `-1`.

```
int iow_i2c_read(char address, char *buf, int size, int
    report_size, int fd);
```

Vgl. die Funktion `iow_i2c_write()` mit dem Unterschied, dass Daten an der Stelle `buf` abgelegt werden. Als Rückgabewert wird im Erfolgsfall die Anzahl gelesener Bytes geliefert, sonst `-1`.

<sup>10</sup> Verfügbar unter `<embedded-linux-dir>/src/tools/`.

```
int iow_i2c_init(int fd);
```

Die Funktion setzt einen IO-Warrior-Baustein in den I<sup>2</sup>C-Modus und liefert die Länge eines Reports zurück. Diese kann für verschiedene Typen variieren und muss für jeden Lese- und Schreibzugriff angegeben werden. Im Fehlerfall wird `-1` zurückgeliefert.

Im folgenden Beispiel wird mithilfe der vorgestellten Routinen ein Temperatursensor vom Typ *Dallas DS1631* mit I<sup>2</sup>C-Adresse `0x4f` ausgelesen (vgl. auch Abschnitt 9.4 im folgenden Kapitel). Der Quelltext ist unter `<embedded-linux-dir>/examples/iic/iic.iowarrior/` verfügbar.

Im Beispiel ist der Temperatursensor an einem IO-Warrior mit Seriennummer `0x00001FEC` angeschlossen.

```
#include <stdio.h>
#include "tools/iowarrior_i2c.h"

int main (int argc, char **argv) {

    char serial[9]      = "00001FEC";
    char basename[20]  = "/dev/iowarrior";
    char buf[2];
    int i;
    char address = 0x4f;

    iow_print_warriors(basename);

    int fd = iow_find_special_descriptor(serial, basename);
    if ( fd < 0 ) {
        printf("Device with serial 0x%s could not be found\n", serial);
        exit(0);
    }

    int report_size = iow_i2c_init(fd);
    buf[0] = 0xac; // command: access config reg
    if (iow_i2c_write(address, buf, 1, report_size, fd) !=1)
        printf("iic write error\n");

    // start conversion
    buf[0] = 0x51; // command: start conversion
    if (iow_i2c_write(address, buf, 1, report_size, fd) !=1)
        printf("iic write error\n");

    while(1) {
        buf[0] = 0xaa; // command: read temperature
        if (iow_i2c_write(address, buf, 1, report_size, fd) !=1)
            printf("iic write error\n");

        if (iow_i2c_read(address, buf, 2, report_size, fd) !=2) //read temp
            registers
            printf("iic read error\n");
        else
            printf("read command byte 0x%0x and 0x%0x\n", buf[0], buf[1]);

        short th = buf[0] << 8;
        short tl = buf[1] & 0x00FF;
        printf("Temperature is: %.1f °C\n", (th | tl) / 250.0);
        sleep(1);
    }
    close(fd);
}
```



Über den Kerneltreiber wird mit den IO-Warrior-Modulen in Form einzelner *Reports* kommuniziert. Die Länge ist je nach verwendetem Modul unterschiedlich und muss für jeden Schreib- und Lesevorgang korrekt angegeben werden. Über die Kerneltreiberfunktion `ioctl(fd, IOW_GETINFO, &info)` wird diese Information innerhalb von `iow_i2c_init()` gelesen und von der `_init()`-Funktion zurückgeliefert. Die genaue Spezifikation des Protokolls und der einzelnen Reports ist im Datenblatt `<embedded-linux-dir>/datasheets/iowarrior24.40.pdf` enthalten. Im Gegensatz zu der in Abschnitt 8.3.2 vorgestellten I<sup>2</sup>C-API kann die Adresse nicht über `ioctl()` für einen oder mehrere Schreib- oder Lesevorgänge festgesetzt werden, sondern muss bei den Schreib- und Lesevorgängen mit angegeben werden. Sollte es notwendig sein, mehr als fünf Bytes Daten zu schreiben oder mehr als sechs Bytes zu lesen, so wird innerhalb von `iow_i2c_read()` und `iow_i2c_write()` die benötigte Anzahl an Reports verarbeitet.

Nach der Übersetzung und dem Start des Programmes sollte eine Meldung ähnlich der folgenden Ausgabe erscheinen (hier ist auch auf die richtige Seriennummer und den korrekten Basisgerätenamen zu achten):

```
$ make
$ sudo ./iic_iowarrior
-----
Found warrior at device /dev/iowarrior0 :
VendorId=07c0
ProductId=1501 (IOWarrior24)
Serial=0x00001FEC
Revision=0x1021
I/O Interface
-----
Found warrior at device /dev/iowarrior1 :
VendorId=07c0
ProductId=1501 (IOWarrior24)
Serial=0x00001FEC
Revision=0x1021
Special Mode Interface
-----
2 Devices found
read command byte 0x1c and 0x70
Temperature is: 29.1 °C
read command byte 0x1d and 0xffffffff0
Temperature is: 30.5 °C
...
```

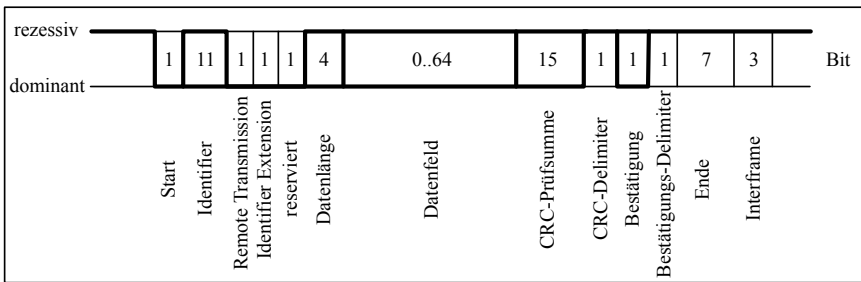
## 8.4 Alternative serielle Bussysteme

In Kapitel 9 wird für den Anschluss externer Komponenten der I<sup>2</sup>C-Bus verwendet. Zuvor sollen aber auch mögliche Alternativen kurz angesprochen werden. Im vorliegenden Abschnitt werden gängige serielle Übertragungsstandards vorgestellt, und es werden die Vor- und Nachteile im Vergleich zu I<sup>2</sup>C erörtert. Die wichtigsten Kenngrößen sind am Ende des Kapitels in Tabelle 8.3 zusammengefasst. Eine Anmerkung: Die RS-232-Schnittstelle ist hierbei nicht aufgeführt; sie wird bereits ausführlich in Kapitel 7 behandelt.

### 8.4.1 Controller Area Network (CAN)

Das *Controller Area Network* (CAN) zählt zur Gruppe der Feldbussysteme und wird zur industriellen Kommunikation verwendet. Ursprünglich wurde der CAN-Bus im Jahre 1983 von der Firma Bosch zur Anbindung von Sensoren und Steuergeräten in der Automobiltechnik entwickelt, um die wachsenden Verkabelungskosten zu senken. Mittlerweile hat fast jedes Auto einen oder mehrere CAN-Busse integriert, an welchen z.B. der Raddrehzahlgeber oder die Lenkwinkelsensoren mit dem ESP-Modul oder dem Zentralsteuergerät kommunizieren. Dieses asynchrone serielle Konzept hat sich in den Jahren nach der Einführung auch in der Prozesstechnik immer mehr durchgesetzt. Ein Hauptgrund hierfür ist die differentielle Übertragung via RS-485 und die damit verbundene hohe Störsicherheit. Weiterhin besitzt der CAN-Bus auch die Möglichkeit der Priorisierung und der Fehlererkennung.

Die Übertragungsraten betragen zwischen 125 kbit/s und 1 Mbit/s, die maximal mögliche Leitungslänge ist umgekehrt proportional und beträgt 40 m für Highspeed-Übertragung und bis zu 500 m bei niedrigster Geschwindigkeit. Ein CAN-Netzwerk ist als Linienstruktur aufgebaut. Die einzelnen Teilnehmer sind jeweils über eine Stichleitung an den Bus angebunden, wobei die Länge der Stichleitung 30 cm nicht überschreiten darf.



**Abb. 8.13.** Aufbau eines CAN-Datentelegramms im Standardformat.

Der CAN-Busstandard besitzt den großen Vorteil, dass Prioritäten für Nachrichten vergeben werden können. Dies ist aufgrund einer bitweisen Arbitrierung möglich. Als Basis dafür dient eine Identifikationsnummer zu Beginn jeder Nachricht (vgl. Abbildung 8.13). Bei gleichzeitigem Senden mehrerer Teilnehmer wird dabei das erste rezessive Bit von einem dominanten Bit überschrieben. Vom unterlegenen Sender wird dies bemerkt und die Nachricht zurückgezogen. So können maximale Übertragungszeiten in bestimmten Grenzen garantiert und entsprechend auch echtzeitfähige Anwendungen umgesetzt werden. Die Nachrichten werden als sog. *Frames* übertragen, wobei außer dem eigentlichen Daten-Frame auch Kontrollstrukturen wie Remote-Frames (zur Anforderung eines Daten-Frames), Error-Frames (signalisieren eine erkannte

Fehlerbedingung) und Overload-Frames (bewirken eine Pause zwischen Daten- und Remote-Frame) auftreten können.

Ein Daten-Frame besteht aus der vorangestellten ID (11 bit im Standardformat oder 29 bit im Extended Format), zusätzlichen Kontrollbits und maximal acht Daten-Bytes (vgl. Abbildung 8.13). Durch die Beschränkung hinsichtlich der Daten-Bytes entsteht vergleichsweise viel Overhead. Der CAN-Busstandard ist daher zur Übertragung größerer Datenmengen weniger gut geeignet.

Das umfangreiche CAN-Protokoll mit den kurzen geforderten Zeitkonstanten würde bei einer Software-Implementierung auf einem Rechner oder Mikrocontroller einen Großteil der Ressourcen in Anspruch nehmen. Entsprechend werden hierfür meist spezielle Schnittstellenbausteine verwendet. In einigen Mikrocontrollern ist eine separate CAN-Einheit bereits integriert. Mit CANopen steht ein abstrakteres Protokoll auf Basis von CAN-Nachrichten zur Verfügung, welches hauptsächlich in der Automatisierungstechnik eingesetzt wird. Vordefinierte Grunddienste wie *Request*, *Indication*, *Response*, *Confirmation* und Kommunikationsobjekte (Service- und Prozessdatenobjekte, Netzwerkmanagement) strukturieren die möglichen CAN-Nachrichten mit zugehörigen IDs und vereinfachen damit die Kommunikation zwischen Geräten unterschiedlicher Hersteller [CiA 08].

#### 8.4.2 Local Interconnect Network (LIN)

Das *Local Interconnect Network (LIN)* wurde als günstige Erweiterung zu CAN für die Kommunikation in Kraftfahrzeugen entwickelt, um intelligente Sensoren über nur eine Leitung anbinden zu können. Entsprechend dient LIN lediglich als Erweiterung, da dieses einfache Konzept nicht an die Sicherheit und Bandbreite des CAN-Busses heranreicht und meist für den Einsatz als Unter-Bus innerhalb einzelner Baugruppen eingesetzt wird.

LIN ist ein Single-Master-Bus, bei dem ein Hauptsteuergerät einen oder mehrere Sensoren wie Türschalter, Drucksensoren im Sitz oder Helligkeitssensoren ausliest. Dafür werden Slave-Bausteine vom Master über definierte Nachrichten zur Datenübertragung aufgefordert. Die Sensordaten bestehen aus maximal acht Daten-Bytes und können bei entsprechender Konfiguration der Filter auch von allen anderen Slaves empfangen werden. Eventuelle Kollisionen entfallen, da ein Master die Kommunikation koordiniert. Die Datenrate ist bei einer maximalen Buslänge von 40 m mit höchstens 19,4 kbit/s wesentlich geringer als bei CAN – für eine Verwendung im Komfortbereich eines Fahrzeuges reicht dies allerdings aus. Das LIN-Protokoll ist byteorientiert, die Übertragung erfolgt asynchron mit nur einer Leitung. Der LIN-Busstandard hat noch nicht die gleiche Verbreitung wie bspw. I<sup>2</sup>C gefunden, da wesentlich weniger Komponenten mit dieser Schnittstelle verfügbar sind. Folglich ist der Einsatz bisher auf den Kfz-Bereich beschränkt.

### 8.4.3 1-Wire-Bus

*1-Wire* ist ein eingetragenes Warenzeichen der Firma *Dallas Semiconductor* und wurde entwickelt, um als günstige Alternative zu I<sup>2</sup>C mit nur einer Datenleitung z. B. Temperatursensoren auszulesen. In vielen Geräten wird 1-Wire verwendet um zwischen Akku und Mikrocontroller zu kommunizieren. Auch im sog. *iButton* sind ROMs, EEPROMs oder Echtzeituhren mit 1-Wire-Schnittstelle versehen (vgl. Abbildung 8.14). Die kompakte und resistente Bauweise in der Edelstahl-Knopfform bietet große Vorteile gegenüber Chipkarten. Werksseitig besitzt jeder *iButton* eine fest einprogrammierte 64-Bit-Adresse, die als Identifikationsnummer für eine Zugangskontrolle verwendet werden kann. Der *ibutton* kann hierzu bspw. am Schlüsselbund getragen werden.



**Abb. 8.14.** Elektronischer Schlüssel *ibutton* mit fest einprogrammierter 64-Bit-Identifikationsnummer.

Bei *1-Wire* handelt es sich ähnlich wie bei LIN um einen Single-Master-Bus, an dem bis zu 100 Slaves angeschlossen werden können. Durch den asynchronen Betrieb kommt dem Zeitverhalten eine besondere Rolle zu: Aufgrund der zulässigen Toleranzen ist bei einer Leitungslänge von bis zu 100 m lediglich ein Betrieb mit maximal 16,3 kbit/s möglich. Eine Besonderheit von 1-Wire-Komponenten ist die Fähigkeit, über die Datenleitung die Versorgungsspannung zu beziehen. Dabei wird während der High-Phase ein interner Kondensator aufgeladen, welcher in der Low-Phase Strom abgibt und den Bedarf für ca. 1 ms überbrücken kann. Somit ist für den Anschluss eines Sensors neben der Datenleitung lediglich eine zweite Leitung als Masseführung notwendig. Aufgrund der geringen Datenrate wird der 1-Wire-Bus in der Automatisierungstechnik bislang selten für den Datenaustausch zwischen Controllern und eher für kleinere Aufgaben wie die Anbindung von Temperatursensoren eingesetzt. Die zulässige Leitungslänge macht ihn aber im Vergleich zu I<sup>2</sup>C auch bspw. als Hausbus interessant.

#### 8.4.4 Serial Peripheral Interface (SPI)

Das *Serial Peripheral Interface* (SPI) wurde von Motorola ins Leben gerufen und ist auch unter dem Begriff *Microwire* bekannt. SPI beschreibt die Hardware-Funktionsweise eines synchronen, unidirektionalen seriellen Bussystems. Von Motorola existiert keine Spezifikation bezüglich des zu verwendenden Protokolls. So können auch Polarität und Phase des Taktsignales für die jeweiligen Geräte unterschiedlich sein. Um Inkompatibilitäten zu vermeiden, muss der steuernde Mikrocontroller individuell konfiguriert werden. Ein Vorteil von SPI ist die Lizenzfreiheit, die maßgeblich zur weiten Verbreitung beigetragen hat.

SPI wird als Single-Master-Bus betrieben. Im Gegensatz zu I<sup>2</sup>C ist ein Betrieb mehrerer Master nicht möglich. Durch die Voll-Duplex-Übertragung, kombiniert mit einer hohen Taktrate bis in den MHz-Bereich, können Datenraten von bis zu 15 Mbit/s erreicht werden. Neben der Taktleitung SCLK werden für beide Richtungen Datenleitungen (MOSI, MISO)<sup>11</sup> sowie eine oder mehrere Slave-Select-Steuerleitungen (SS) benötigt. Mithilfe dieser Leitungen signalisiert der Master die Auswahl eines Slave-Bausteines. Wird nur ein Slave am Bus betrieben, so kann unter Umständen auf diese Leitung verzichtet werden. Diese Möglichkeit hängt aber von der spezifischen Funktionsweise des Slave ab. Nach der Auswahl erfolgt synchron zum Taktsignal des Masters die Datenübertragung.

SPI wird hauptsächlich für die Kommunikation zwischen Mikrocontrollern innerhalb eines Systems oder als Schnittstelle für das sog. *In-System-Programming* (ISP) eingesetzt. Eine bekannte SPI-Komponente ist die MMC-Speicherkarte, die in Geräten wie Digitalkameras oder Mobiltelefonen Anwendung findet. Ähnlich wie für I<sup>2</sup>C, so existiert auch für SPI eine große Zahl an Peripheriebausteinen in Form von AD-Wandlern, LCD-Treibern oder Temperatursensoren. Da keinerlei Störschutz vorgesehen ist, ist die Komponentenanbindung über Strecken von mehr als einem Meter nicht zu empfehlen. Wie auch bei I<sup>2</sup>C kann allerdings die Verlangsamung der Taktrate die Störsicherheit und damit auch die maximale Übertragungstrecke erhöhen. Weiterhin können die Pegel aufgrund der unidirektionalen Übertragung auch leicht auf den symmetrischen RS-485-Standard gewandelt werden.

#### 8.4.5 Universal Serial Bus (USB)

Der *Universal Serial Bus* (USB) wurde im Jahre 1996 von der Firma Intel entwickelt, um im PC-Bereich die wachsende Anzahl von Peripheriegeräten (Maus, Tastatur, Drucker, Speichersticks, WLAN-Karten usw.) mit einer einheitlichen Schnittstelle zu versehen. Dadurch hat sich die Verwendung von

---

<sup>11</sup> Master Output – Slave Input bzw. Master Input – Slave Output.

RS-232, PS/2, der parallelen Druckerschnittstelle, des Gameport oder auch von PCMCIA heutzutage für viele Aufgaben erübrigt. Eine Erweiterung der zunächst in der Spezifikation 1.0 definierten Geschwindigkeit von maximal 12 Mbit/s erfolgte im Jahre 2000 in der Version 2.0 auf 480 Mbit/s. Dadurch wurde USB auch als schnelle Schnittstelle für externe Festplatten, DVD-Laufwerke und Videogeräte interessant und trat in direkte Konkurrenz zu dem von Apple und Sony im Jahr 1995 entwickelten 1394-Firewire-Bus.

Auch wenn es der Name vermuten lässt, so realisiert USB keine Linienstruktur mit Stichleitungen, sondern eine Baumstruktur. Die Verzweigungen entstehen durch die Verwendung sog. *Hubs*.<sup>12</sup> Am Kopf der Baumstruktur sitzt ein zentraler Host-Controller, welcher nur einmal am Bus existiert und die Kommunikation mit den bis zu 127 Slaves koordiniert. Der Host-Controller identifiziert ein angestecktes Gerät und lädt automatisch den notwendigen gerätespezifischen Treiber. Für bestimmte Geräteklassen existieren generische Treiber, sodass diese Geräte direkt nach dem Einstecken funktionsbereit sind (vgl. Tabelle 8.2). Herstellerspezifische Geräte verlangen die Installation separater Treiber. Für den Anschluss der Geräte sieht die USB-Spezifikation auch das sog. *Hot Plugging*, also das Hinzufügen und Entfernen von Geräten im laufenden Betrieb vor.

Basisklasse	Beschreibung
00h	Platzhalter für universelle Gerätebeschreibung
01h	Audio
02h	Kommunikation und CDC-Kontrolle
03h	HID (Human Interface Device)
05h	PID (Physical Interface Device)
06h	Bilder
07h	Drucker
08h	Massenspeicher
09h	Hub
0Ah	CDC-Daten
0Bh	Chipkartenlesegerät
0Dh	Content Security
0Eh	Video
0Fh	Gesundheitsbereich
DCh	Diagnosegerät
E0h	kabelloser Controller
EFh	Verschiedenes
FEh	anwendungsabhängig
FFh	herstellerabhängig

**Tabelle 8.2.** USB-Geräteklassen nach Spezifikation 2.0.

<sup>12</sup> Engl. für Drehkreuz.

Die Datenübertragung erfolgt symmetrisch bidirektional über ein verdrehtes Leitungspaar, um Gleichtaktstörungen zu unterdrücken und die hohe Übertragungsrate zu ermöglichen. Zusätzlich zu den beiden Datenleitungen wird eine Versorgungsleitung V+ samt Masse mitgeführt, um Endgeräte mit geringem Stromverbrauch direkt über die USB-Leitung versorgen zu können. Je nach Controller bzw. Hub stehen für die Versorgung 100 mA (*Low Power*) oder 500 mA (*High Power*) zur Verfügung. Diese Limitierung ist eine häufige Ursache für auftretende Kommunikationsprobleme und muss beim Anschluss mehrerer Geräte berücksichtigt werden. Eine Verwendung aktiver Hubs kann helfen, die Versorgungsleistung auf dem Bus zu erhöhen. Die maximale Leitungslänge zwischen Host bzw. Hub und Gerät ist laut Spezifikation auf 5 m begrenzt. Eine Verlängerung ist durch Verwendung mehrerer Hubs oder eines aktiven USB-Verlängerungskabels möglich.

Für die Realisierung einer USB-Schnittstelle sind spezielle USB-Controller-Bausteine notwendig. Ein Beispiel hierfür ist der National Semiconductor USBN9603/USBN9604, der die Anbindung an einen 8-Bit-Datenbus realisiert. Weiterhin existieren auch Mikrocontroller wie der Atmel AT98C5131/AT98C5132, bei welchen eine USB-Einheit bereits integriert ist. Die USB-Anbindung ist auf Host-Seite vergleichsweise komplex und umfasst auch die Programmierung eines eigenen Gerätetreibers. Dadurch und auch aufgrund der eingeschränkten Übertragungsstrecke ist eine USB-Verbindung zwischen einem eingebetteten System und dessen Sensor- und Aktorperipherie nicht empfehlenswert. Wird eine Erweiterung hinsichtlich IO-Pins oder I<sup>2</sup>C benötigt, so kann auch auf die kommerziellen Fertigbausteine der Firma Code Mercenaries ausgewichen werden (vgl. Abschnitt 7.5). Durch die fehlende Echtzeitfähigkeit und die kurzen Übertragungsstrecken kommt USB als Ersatz für einen Feldbus kaum in Frage. Der Standard ist aber auch für eingebettete Systeme ideal geeignet, externen Speicher, Festplatten oder Kameras anzubinden oder das System um weitere Schnittstellen wie WLAN, Bluetooth oder Audio zu erweitern (vgl. Kapitel 10).

Unter Linux werden ab Kernel-Version 2.4 UHCI-, OHCI- und EHCI-Controller<sup>13</sup> unterstützt. Für Kernel 2.6 wurden die Treiber weiter verbessert, entsprechend sollten auf allen verfügbaren vollwertigen Linux-Distributionen bereits die Basistreiber für die Verwendung von USB-Geräten vorhanden sein.

---

<sup>13</sup> Das *Universal Host Controller Interface (UHCI)* und das *Open Host Controller Interface (OHCI)* stellen Unterstützung für USB-1.1-Geräte zur Verfügung. Das später entwickelte *Enhanced Host Controller Interface (EHCI)* wickelt Übertragungen im Highspeed-Modus mit 480 Mbit/s ab. Weitere Informationen zu diesen Standards finden sich im USB-Entwicklerforum unter <http://www.usb.org>.

Bezeichnung	Übertragung	Topologie	Datenrate	Länge	Teilnehmer	Besonderheiten
I <sup>2</sup> C	synchron, halb-duplex	Linienstruktur	100 kbit/s – 3,4 Mbit/s	2 m – 3 m > 100 m mit Expander	112	günstige Komponenten
CAN	asynchron, Frame-orientiert	Linienstruktur	125 kbit/s – 1 Mbit/s	500 m – 40 m	bis 128	Hohe Übertragungs- sicherheit, Echtzeitfähigkeit
LIN	asynchron, seriell, Frame-orientiert	nicht vorgegeben	20 kbit/s	40 m	16	–
1-Wire	asynchron	Linienstruktur	15 kbit/s – 125 kbit/s	100 m – 300 m	16	Stromversorgung über Datenleitung
SPI	synchron, voll-duplex	Stern- und Linienstruktur	bis 15 Mbit/s	1 m	4096, aber je eine Leitung SS	–
USB	asynchron, halb-duplex	Stern- und Linienstruktur	1,5 Mbit/s – 480 Mbit/s	5 m	127	schnell, eignet sich zur Anbindung von Speichermedien

**Tabelle 8.3.** Vergleich verschiedener serieller Buskonzepte. I<sup>2</sup>C reicht für viele Anwendungen im Hinblick auf Übertragungslänge und Störuneempfindlichkeit aus und ist zudem einfach zu implementieren. I<sup>2</sup>C-Komponenten sind aufgrund der Verwendung im Consumer-Bereich preisgünstig zu beziehen.



## Inter-IC-Bus-Komponenten

### 9.1 Einführung

Nachdem im vorigen Kapitel die Grundlagen des I<sup>2</sup>C-Busses erklärt und einfache Routinen für Lese- und Schreibzugriffe vorgestellt wurden, soll nun in diesem Kapitel auf die Ansteuerung spezieller I<sup>2</sup>C-Bausteine eingegangen werden. Die zu diesem Zweck vorgestellten Kompletต์module sind günstig über diverse Elektronikversandhäuser<sup>1</sup> zu beziehen, können aber mit etwas Elektro- nikerfahrung auch selbst aufgebaut werden. Anhand einfacher Beispiele wird die Verwendung folgender I<sup>2</sup>C-Bausteine erklärt:

- Philips PCF8574: 8-Bit-I/O-Erweiterung
- Dallas DS1631: Temperatursensor
- Philips PCF8591: 8-Bit-D/A- und A/D-Wandler
- TMC222: Schrittmotortreiber
- AT24Cxx: Serielle EEPROM-Chipkarte
- Philips PCA9548A: 8-Kanal-I<sup>2</sup>C-Multiplexer

Um Verwirrungen vorzubeugen, wird im Folgenden für I<sup>2</sup>C-Geräte die Bezeichnung *Modul* verwendet, für I<sup>2</sup>C-ICs die Bezeichnung *Baustein*. Sämtliche in diesem Kapitel verwendeten Module und Bausteine sind mit Bezugsquellen in Anhang E aufgelistet. Jeder der genannten Bausteine benötigt unterschiedliche Routinen zur Ansteuerung. Entsprechend liegt der Schluss nahe, eine I<sup>2</sup>C-Klassenbibliothek zu erstellen und die Methoden darin zu sammeln. Die Bibliothek wird in der Programmiersprache C++ umgesetzt, um Eigenschaften wie Vererbung nutzen zu können und eine gute Strukturierung zu ermöglichen. Es wird davon ausgegangen, dass der Leser mit den Grundlagen der C++-Programmierung vertraut ist – eine gute Einführung bietet [Stroustrup 00].

---

<sup>1</sup> Vgl. die Komponentenliste in Tabelle E.2.

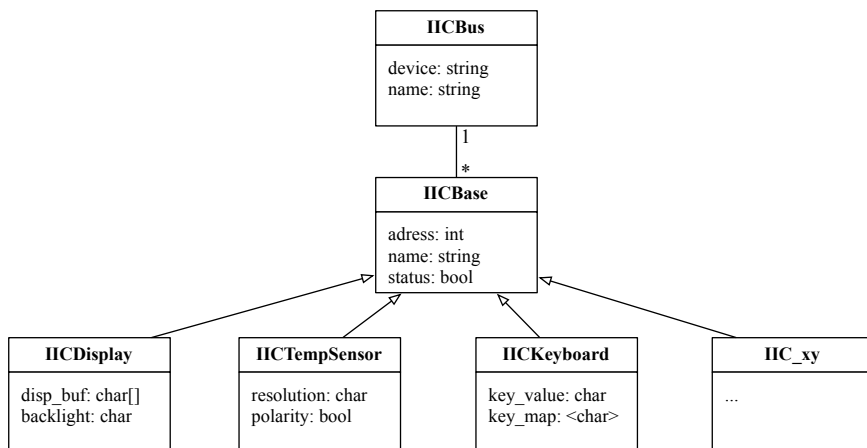
Um eine geeignete Klassenhierarchie für diese Bibliothek zu entwerfen, ist es sinnvoll, zunächst die Randbedingungen und Anforderungen zu klären. Folgende Vorgaben werden bei der Spezifikation der Bibliothek zugrunde gelegt:

- Es existiert genau ein Bus-Objekt für jede vorhandene I<sup>2</sup>C-Schnittstelle (z.B. `/dev/i2c-0`). Nur dieses Bus-Objekt selbst darf auf diese Gerätedatei zugreifen.
- Jedes Bus-Objekt verwaltet und überwacht die zugeordneten Module und stellt Diagnosefunktionen bereit.
- Ein Modul ist genau einem Bus zugeordnet.
- Alle Module besitzen eine I<sup>2</sup>C-Adresse und sollen zu- und abschaltbar sein.
- Jedes Modul benötigt einheitliche Methoden zur I<sup>2</sup>C-Ansteuerung und spezielle Methoden in seiner Benutzerschnittstelle.
- Module können über einen I<sup>2</sup>C-Multiplexer-Baustein an einen I<sup>2</sup>C-Unterbus angeschlossen werden. Das Bus-Objekt muss davon Kenntnis besitzen.

Die Bereitstellung der Methoden zur Verwaltung und Überwachung der I<sup>2</sup>C-Schnittstelle soll in einer eigenen Klasse vom Typ `IICBus` stattfinden. In der Praxis sind mehrere I<sup>2</sup>C-Module genau einem Bus-Objekt zugeordnet, sodass eine Ableitung der Modulklassse von `IICBus` wenig sinnvoll erscheint. Zur Bereitstellung spezifischer Methoden in der API wird für jedes Modul eine eigene Klasse erstellt (Beispiele: Ausgabe einer Zeichenkette auf ein I<sup>2</sup>C-Display, Einlesen der Temperatur eines I<sup>2</sup>C-Sensors). Allerdings wurden auch einige Gemeinsamkeiten zwischen Modulen identifiziert: Alle Module besitzen eine I<sup>2</sup>C-Adresse und benötigen Zugriffsfunktionen auf ein Bus-Objekt. Eine einheitliche Modul-Basisklasse könnte diese Gemeinsamkeiten kapseln und davon abgeleitete Unterklassen die spezifischen Methoden umsetzen.

Die Ansteuerung der verwendeten I<sup>2</sup>C-Bausteine ist nicht sehr komplex, sodass eine nochmalige Unterteilung der Modul-Methoden hin zu Baustein-Methoden kaum lohnenswert erscheint. Auf eine weitere Vererbung der verschiedenen Bausteintypen soll deshalb verzichtet werden – eine zu feingranulare Einordnung bringt nicht immer eine bessere Übersichtlichkeit mit sich. Abbildung 9.1 zeigt die Klassenhierarchie, wie sie für die I<sup>2</sup>C-Bibliothek verwendet werden soll.

In den folgenden Abschnitten wird die Entwicklung der I<sup>2</sup>C-Bibliothek und die Ergänzung um Modulklassen Schritt für Schritt erklärt. Abschnitt 9.8 beschreibt die Aufteilung in mehrere I<sup>2</sup>C-Unterbusse, um eine Erweiterung des Adressbereiches zu erreichen, falls mehr als acht Geräte gleichen Typs betrieben werden sollen (üblicherweise sind nur drei Adressbits frei wählbar).



**Abb. 9.1.** Klassenhierarchie der I<sup>2</sup>C-Modulbibliothek als UML-Diagramm: Zwischen IICBus und IICBase besteht eine Assoziation. Mehrere Basis-Objekte sind dabei genau einem Bus-Objekt zugeordnet (Kardinalität). Die Basisklasse wird durch weitere Modulklassen spezialisiert, um besondere Funktionalitäten umzusetzen.

## 9.2 Die I<sup>2</sup>C-Bibliothek

Die vollständige I<sup>2</sup>C-Klassenbibliothek befindet sich im Verzeichnis `<embedded-linux-dir>/src/iic/`. Zahlreiche I<sup>2</sup>C-Anwendungen im Beispielerzeichnis `<embedded-linux-dir>/examples/iic/` veranschaulichen die Verwendung der jeweiligen Modulklassen.

### 9.2.1 Die Klasse IICBus

Die Klasse IICBus bildet die Basis für den Zugriff auf die I<sup>2</sup>C-Schnittstelle und dient als Ausgangspunkt für die Entwicklung weiterer Modulklassen. Zunächst wird in der Klasse ein Strukturtyp `modinfo_t` definiert, in welchem Informationen über ein angeschlossenes I<sup>2</sup>C-Modul enthalten sind. Diese beinhalten einen Zeiger `p_module` auf das Basisobjekt des Moduls, dessen I<sup>2</sup>C-Adresse `addr`, einen Zeiger `p_mux` auf ein Multiplexer-Objekt (falls das Modul an einem Unterbus angeschlossen ist, sonst NULL) sowie den verwendeten Kanal `mux_channel`.

```

typedef struct {
    IICBase* p_module;    // pointer to base object
    int addr;            // i2c address
    IICMultiplexer* p_mux; // pointer to multiplexer object
    int mux_channel;      // connected multiplexer channel
} t_modinfo;
  
```

Weiterhin besitzt jeder Bus einen Namen `m_name` zur Beschreibung und muss den Filedescriptor `m_fd` auf die zugeordnete I<sup>2</sup>C-Schnittstelle speichern. Ein I<sup>2</sup>C-Anschluss kann entweder durch eine im System vorhandene Schnittstelle oder aber über einen IO-Warrior-Baustein realisiert werden. Diese Eigenschaft wird in `m_iface_type` hinterlegt. Kommt ein IO-Warrior-Baustein zum Einsatz, so muss die jeweilige Länge `m_iow_repsize` eines *Reports* beachtet werden und wird deshalb ebenfalls abgelegt. Ein Objekt der Klasse `IICBus` soll abstrakte Module vom Typ `IICBase` verwalten und muss dafür keine speziellen Modulinformationen besitzen. Die allgemeinen Modulinformationen werden anhand einer Identifikationsnummer im Container `m_modules` abgelegt. Letztendlich muss ein Bus-Objekt von der aktuellen Verzweigung in einen Unterbus Kenntnis besitzen und diese in Form eines Zeigers `mp_curr_mux` auf den aktiven Multiplexer und dessen ausgewählten Kanal `m_curr_channel` vorhalten. Mit Ausnahme von `IICMultiplexer` sind dem `IICBus` keine weiteren Spezialisierungen der Modul-Basisklasse bekannt. Um die Klasse *thread safe* zu machen, werden alle Methoden, die einen Zugriff auf den I<sup>2</sup>C-Bus enthalten, mit einem Mutex `m_io_mutex` geschützt. Die Methoden sind damit atomar und nicht unterbrechbar. In Kapitel 12 wird dieses Thema nochmals ausführlich behandelt.

Zusammenfassend werden für Objekte vom Typ `IICBus` folgende Membervariablen benötigt:

```
string m_name;           // bus name
int m_fd;                // device file descriptor (e.g. /dev/i2c-0)
int m_iface_type;        // 0 = "native"; 1 = "iowarrior"
int m_iow_repsize;       // hold size of IOWarrior reports
map<int, modinfo_t> m_modules; // map to hold pairs id and device info
IICMultiplexer* mp_curr_mux; // active multiplexer at this time
int m_curr_channel;      // current channel for active mux
Mutex m_io_mutex;        // protect iic bus access
```

Als Schnittstelle zur eigenen Anwendung stehen dem Anwender folgende Funktionen zur Verfügung:

```
IICBus(string name, int iface_type, string iow_serial);
```

Der Konstruktor erzeugt ein Objekt vom Typ `IICBus` unter Angabe eines Namens `name`, des Schnittstellentyps `iface_type` (=0 für I<sup>2</sup>C-Schnittstellen der Form `/dev/i2c-0`; =1 für I<sup>2</sup>C-Erweiterungen über IO-Warrior-Bausteine) und einer IO-Warrior-Seriennummer `iow_serial`. Diese dient der Unterscheidung mehrerer IO-Warrior-Platinen.

```
string getName();
```

Liefert die Bezeichnung des Bus-Objektes.

```
void init();
```

Initialisiert alle in `m_modules` eingetragenen Geräte, indem deren Initialisierungsroutinen aufgerufen werden.

```
int autodetect();
```

Versucht, angeschlossene Geräte durch Lesevorgänge im Adressbereich 1–127 zu finden und schreibt identifizierte Adressen auf die Standardausgabe. Als Rückgabewert dient die Anzahl der gefundenen Geräte. Diese Möglichkeit funktioniert für viele, jedoch leider nicht für alle I<sup>2</sup>C-Bausteintypen.

```
int addModule(int addr, IICBase* module);
```

Fügt der Liste `m_modules` ein weiteres Modul hinzu. Benötigt werden die I<sup>2</sup>C-Adresse `addr` und ein Zeiger `module` auf das Basisobjekt. Als Rückgabewert wird eine eindeutige Identifikationsnummer vom Typ `int` geliefert. Der Aufruf erfolgt im Konstruktor der Klasse `IICBase`.

```
void rmModule(int id);
```

Löscht ein Modul mit Identifikationsnummer `id` aus der Liste `m_modules`.

```
void makeSubModule(IICBase& module, IICMultiplexer& mux, int channel);
```

Kennzeichnet ein Modul `module` als Teilnehmer an einem I<sup>2</sup>C-Unterbus und ordnet diesem einen Multiplexer-Baustein `mux` mit Kanal `channel` zu.

```
void printModules();
```

Gibt eine Liste aller angemeldeten Module mit weiteren Informationen auf der Standardausgabe aus.

```
int iicWrite(int id, char* buf, int num);
```

Schreibt `num` Daten-Bytes von Quelle `buf` an das Modul mit Identifikationsnummer `id`. Die zugehörige I<sup>2</sup>C-Adresse wird automatisch bestimmt. Falls notwendig wird auf den entsprechenden Unterbus umgeschaltet und je nach eingestellter Schnittstelle mit einem nativen I<sup>2</sup>C-Anschluss oder einer IO-Warrior-Platine kommuniziert. Im Erfolgsfall wird als Rückgabewert die Anzahl geschriebener Daten-Bytes geliefert, im Fehlerfall der Wert `-1`.

```
int iicRead(int id, char* buf, int num);
```

Wie `iicWrite()`, jedoch werden die Daten an der Stelle `buf` abgelegt.

Die als `private` deklarierte Methode `checkConfig()` dient dazu, unmittelbar vor Initialisierung der einzelnen Module zunächst die in `m_modules` hinterlegten Modulinformationen auf Gültigkeit zu überprüfen: Es dürfen keine I<sup>2</sup>C-Bausteine mit identischer Adresse an einem Unterbus existieren. Im Fehlerfall wird die Anwendung beendet.

### 9.2.2 Die Klasse `IICBase`

Die generische Klasse `IICBase` soll modulübergreifende Informationen und Methoden enthalten und als Basis für spezielle Unterklassen dienen. Jedes Modul

besitzt eine I<sup>2</sup>C-Adresse `m_addr`, eine eindeutige Identifikationsnummer `m_id`, und einen Namen `m_name`. Weiterhin ist der Status des Moduls in `m_status` gespeichert. Die Information, ob es sich um ein entnehmbares Modul handelt, ist in `m_is_removable` abgelegt. Letzteres hilft, unnötige Fehlermeldungen bei I<sup>2</sup>C-Chipkarten oder ähnlichen Modulen zu vermeiden. Auftretende Lese- oder Schreibfehler werden in der Variablen `m_err_count` mitgezählt. Eine Referenz `m_bus` wird benötigt, um Schreib- und Lesezugriffe über das zuständige Bus-Objekt ausführen zu können.

Zusammengenommen besitzt die Klasse `IICBase` folgende Membervariablen:

```
int      m_addr;           // i2c address
int      m_id;            // module id (for bus identification)
string   m_name;          // module name
bool     m_status;        // module status (true/false)
unsigned char m_is_removable; // indicates removable device (chipcard)
int      m_err_count;     // counts read/write failures
IICBus& m_bus;            // reference to connected iicbus
```

Diese Daten sind für Unterklassen nicht von Interesse und werden deshalb als `private` deklariert. Für die API sind folgende Klassenmethoden relevant:

`IICBase(IICBus& bus, int addr, string name);`

Erzeugt ein Objekt vom Typ `IICBase` und erhält als Argumente eine Referenz `bus` auf das Bus-Objekt mit I<sup>2</sup>C-Interface, die I<sup>2</sup>C-Adresse `addr` des Bausteins und eine Bezeichnung `name`. Der Name kann beliebig gewählt werden und dient nur dem Anwender als beschreibendes Element.

`string getName();`

Liefert den Namen des Moduls zurück.

`int getErrCount();`

Dient zur Abfrage des Fehlerspeichers und liefert die Anzahl bislang aufgetretener Fehler zurück. Diese werden durch fehlgeschlagene Lese- oder Schreiboperationen verursacht.

`bool getStatus();`

Liefert den aktuellen Status des Moduls zurück (`true=An`, `false=Aus`).

`void setStatus(bool status);`

Setzt den Status des Moduls auf `true` oder `false`. Bei abgeschaltetem Modul werden keine Lese- oder Schreiboperationen auf den Bus durchgeführt.

`void setIsRemovable(unsigned char rem);`

Setzt die Eigenschaften des Moduls auf *entfernbar* und verhindert damit die Fehleranhäufung und eine etwaige Abschaltung.

`int writeData(char* buf, int num);`

Schreibt bei aktiviertem Modul `num` Daten-Bytes von der Quelle `buf` auf das Objekt `IICBus`. Beim Auftreten von insgesamt `MAXERR` Fehlern wird

das Modul abgeschaltet. Damit werden weitere Lese- oder Schreibzugriffe unterbunden. Im Erfolgsfall wird die geschriebene Anzahl Daten-Bytes zurückgeliefert, sonst  $-1$ .

```
int readData(char* buf, int num);
```

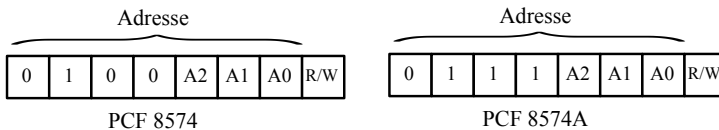
Wie `writeData()`, die Daten werden jedoch gelesen und an Stelle `buf` abgelegt.

Nachdem zunächst der grundlegende Aufbau der Bibliothek erläutert wurde, soll nun mit weiteren Geräteklassen die Funktionsweise und Einbindung von I<sup>2</sup>C-Bausteinen veranschaulicht werden. Die nachfolgenden Klassen werden von `IICBase` abgeleitet.

## 9.3 Tastatur- und LC-Display-Ansteuerung mit PCF8574

### 9.3.1 Philips 8-Bit-I/O-Erweiterung PCF8574

Beim PCF8574-Baustein handelt es sich um eine 8-Bit-I/O-Erweiterung für den I<sup>2</sup>C-Bus. Der Chip wird mit 5V betrieben und liefert einen Strom von 20 mA je Ausgang. Hiermit können bspw. LEDs direkt angesteuert werden. Es stehen die zwei Varianten 8574 und 8574A zur Verfügung, die sich durch unterschiedliche hartkodierte Adressmuster auszeichnen (vgl. Abbildung 9.2). Die Adresse kann jeweils über drei Pins eingestellt werden, sodass insgesamt bis zu acht Bausteine eines Typs an einem (Unter-)Bus betrieben werden können.



**Abb. 9.2.** Adresseinstellungen der Bausteine PCF8574 und PCF8574A.

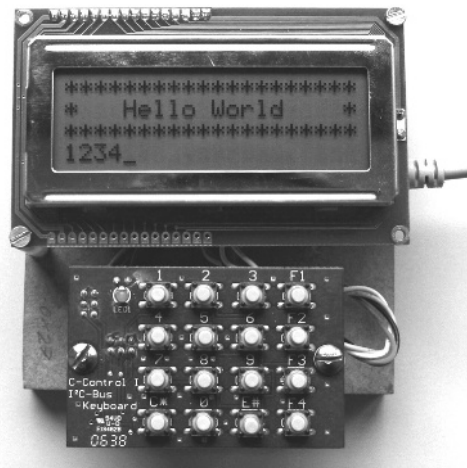
Die Ansteuerung ist denkbar einfach: Da der I<sup>2</sup>C-Bustreiber das Protokoll abhandelt und die richtige Bausteinadresse mit passendem R/W-Bit jeder Nachricht voranstellt, können die Daten-Bytes direkt gesendet bzw. eingelesen werden (vgl. hierzu auch die Beispiele in Kapitel 8).

Dabei ist für diesen Baustein jeweils nur ein Byte zu transferieren, welches den PCF8574-Anschlüssen P0 bis P7 entspricht. Durch Lesen oder Schreiben eines Bytes wird direkt zwischen den Übertragungsmodi umgeschaltet. Die Übertragung erfolgt quasi-bidirektional – die Richtung muss also nicht über

ein separates Register festgelegt werden.<sup>2</sup> Die als Eingänge verwendeten Pins sollten vor dem Lesen auf HIGH gesetzt sein.

Neben den acht I/O-Leitungen stellt der PCF8574 zusätzlich eine Interruptleitung zur Verfügung. Tritt an den Eingängen (im Lesemodus, d. h. nach einem Lesevorgang) eine steigende oder fallende Flanke auf, so wird ein Interrupt an Leitung INT ausgelöst. Durch den nächsten Bausteinzugriff wird die Interruptschaltung wieder reaktiviert. Für die Auswertung wäre ein weiterer I/O-Port am Router notwendig. Da dies nicht zum Buskonzept passt, soll hier auf eine Auswertung der Interrupts verzichtet werden.

Als Beispiel wird in diesem Abschnitt die Ansteuerung eines Bedienpanels gezeigt. Der Quelltext ist im I<sup>2</sup>C-Beispielverzeichnis unter **panel** zu finden. Das Panel besteht aus einer 4×4-Matrixtastatur und einer Adapterplatine zum Anschluss eines LC-Displays auf Basis eines Hitachi HD44780-Controllers (vgl. Komponentenliste in Anhang E.2). Abbildung 9.3 zeigt beide Module.



**Abb. 9.3.** 4×4-Matrixtastaturmodul (Conrad) und LCD-Adapter (CC-Tools) mit Display.

Zur Stromversorgung über die mitgeführten 5 V des I<sup>2</sup>C-Anschlusses vgl. auch die Steckerbelegung entsprechend Abbildung 8.11 im vorangegangenen Kapitel.

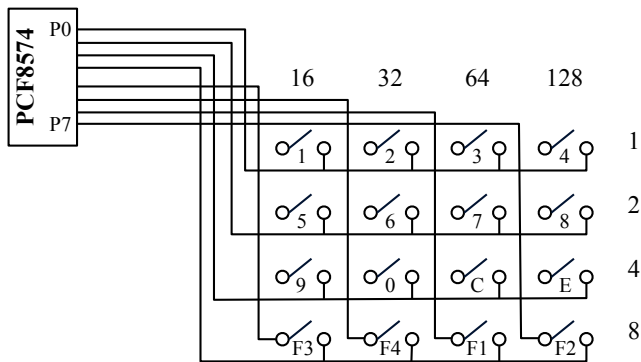
<sup>2</sup> Um Überraschungen zu vermeiden ist beim Entwurf einer eigenen Schaltung zu berücksichtigen, dass die Ausgänge des PCF8574 nach dem Einschalten zunächst auf HIGH gelegt sind.



### 9.3.2 I<sup>2</sup>C-Tastaturmodul

Wenn am System eine USB-Schnittstelle vorhanden ist, so kann üblicherweise auch eine USB-Tastatur verwendet werden. In Einzelfällen kann es jedoch vorkommen, dass eine Eingabe dezentral aus einer Entfernung von mehr als fünf Metern vom Rechnersystem erfolgen soll. In diesem Fall ist die Integration eines Tastaturmoduls in das I<sup>2</sup>C-Buskonzept wünschenswert.

Die I<sup>2</sup>C-Bus-Tastatur kann eine 4×4-Matrix von 16 Tastern abfragen. Die drei Leitungen zur Konfiguration der Adresse sind auf Jumper herausgeführt. Eine Anordnung in Matrixform gemäß Abbildung 9.4 erfordert zwei separate Abfrageschritte, bietet aber folgende Vorteile: Während bei Belegung der Taster mit einem festen Potential von GND oder 5 V und dem Einlesen über I/O-Leitungen nur der Anschluss von acht Tastern möglich wäre, so können durch diese Form der Abfrage bis zu 16 Taster ausgelesen werden.



**Abb. 9.4.** Schaltplan eines PCF8574-Bausteins mit 16 angeschlossenen Tastern im C-Control-Modul I<sup>2</sup>C-Bus-Tastatur von Conrad. Die Wertigkeit der einzelnen Zeilen und Spalten ist über der Matrix und seitlich davon abgedruckt.

Die 5 V-Spannung, mit der ein Tastendruck über eine Datenleitung identifiziert wird, ist über eine andere I/O-Leitung zu generieren. Dann kann durch Anlegen einer Spannung auf P0–P3 in der Folge über P4–P7 gelesen werden, welche Zeile der Matrix momentan HIGH ist. Hierbei können auch Taster in mehreren Zeilen gedrückt sein. Durch Anlegen einer Spannung auf P4–P7 kann in einem zweiten Schritt die zugehörige Spalte identifiziert werden.

### 9.3.3 Die Klasse IICKeyboard

Die Klasse `IICKeyboard` soll Geräte vom Typ I<sup>2</sup>C-Bus-Tastatur repräsentieren. Da ein interruptgesteuertes Lesen nicht möglich ist, muss zum Einlesen eine

aktive Abfrage per sog. *Polling* erfolgen. Die Tastatur besitzt keinen Puffer, um Zeichen zwischenspeichern. Nur durch eine rechtzeitige Abfrage können alle Eingaben gelesen werden. Im Regelfall ist bekannt, wann eine Benutzereingabe erfolgen soll. Entsprechend kann eine blockierende Methode `getKey()` realisiert werden, welche nach einem Tastendruck das zugeordnete ASCII-Zeichen zurückgibt. Naheliegender aber nicht zwingend ist die Rückgabe der aufgedruckten Zeichen. Die Zuordnung zwischen Zahlenwerten und Zeichen wird in einer Tabelle vom Typ `map<int, char>` abgelegt. Als `key` dient die Summe der in beiden Abfrageschritten gelesenen Zeilen- und Spaltenwerte. Um ein Prellen der Taster zu vermeiden, muss der bisherige Zahlenwert in jedem Abfrageintervall gespeichert werden. Als Membervariablen werden entsprechend benötigt:

```
char m_buf; // r/w buffer
unsigned char m_key_val, m_key_val_old; // current and prev key values
map<unsigned char, unsigned char> m_keymap; // value-char pairs
```

Die Erstellung der Einträge in `m_keymap` wird in der Initialisierungsroutine vorgenommen. Über das Makro `POLL_CYCLE` wird die Periodendauer für ein Polling-Intervall in Mikrosekunden festgelegt. Die überschaubare API beinhaltet lediglich die folgenden Funktionen:

```
IICKeyboard(IICBus& bus, int addr, string name);
```

Erzeugung eines Objektes vom Typ `IICKeyboard` unter Angabe der Referenz auf das Objekt `bus`, der I<sup>2</sup>C-Adresse `addr` und einer Bezeichnung `name`.

```
void init();
```

Vorbelegung der Tabelle `m_keymap` mit Paaren der Form `<wert, zeichen>`.

```
char getKey();
```

Bestimmung eines Schlüssels aus aktiver Zeile und Spalte. Als Rückgabewert dient das in `m_keymap` zugeordnete Zeichen.

### 9.3.4 I<sup>2</sup>C-LC-Display

Vielen Embedded-Systemen fehlt die Möglichkeit, über VGA einen Bildschirm anzuschließen. Weiterhin ist abhängig von der verwendeten Embedded-Linux-Distribution u. U. generell keine grafische Anzeige möglich. Oft ist aber zumindest eine (dezentrale) Textausgabe wünschenswert, wie sie durch den Aufbau eines Terminals aus Anzeige und Tastatur möglich wird. Eine Eingliederung in das I<sup>2</sup>C-Konzept erscheint deshalb sinnvoll. Ein LC-Display<sup>3</sup> bietet zumindest die Möglichkeit, einfache Daten wie bspw. den System- oder Anwendungsstatus auszugeben.

LC-Displays können als zeichenorientierte (alphanumerische) oder grafische Anzeigen ausgeführt sein, wobei die grafischen Varianten teurer und

<sup>3</sup> Engl. *Liquid Crystal Display*, Flüssigkristallanzeige.

aufwändiger in der Ansteuerung sind. Im Hinblick auf diese Nachteile grafischer LCDs und auf die begrenzte Bandbreite des I<sup>2</sup>C-Busses scheint ein alphanumerisches LC-Display die bessere Lösung. Bei dieser Form der Anzeige ist ein Zeichensatz fest im Speicher des Display-Controllers hinterlegt. In der Regel stehen ein, zwei oder vier Zeilen mit jeweils 16 bis 20 Zeichen zur Verfügung.

Weiterhin ist anzumerken, dass fast alle LC-Displays von einem Hitachi-HD44780-Controller oder einem kompatiblen Derivat angesteuert werden. Dieser Baustein hat sich als Quasi-Standard durchgesetzt.

### 9.3.5 LC-Display-Treiberbaustein HD44780

Der HD44780<sup>4</sup> besitzt einen 80 Zeichen großen Textpuffer und wird über ein paralleles Interface angesteuert. Je nach verwendetem Modus werden vier oder acht Datenleitungen und drei Steuerleitungen benötigt. Während die Ansteuerung im 8-Bit-Modus etwas einfacher ist, kommt der Betrieb im 4-Bit-Modus mit nur sieben I/O-Pins aus – für viele Anwendungen ist dies ein großer Vorteil. Am PCF8574 stehen insgesamt nur acht Bit zur Verfügung, sodass die Wahl auf den 4-Bit-Modus fällt. Das letzte zur Verfügung stehende Bit kann zur Ansteuerung einer Hintergrundbeleuchtung verwendet werden.

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Beschreibung
0	0	0	0	0	0	0	0	0	1	Clear Display
0	0	0	0	0	0	0	0	1	*	Return Home
0	0	0	0	0	0	0	1	I/D	S	Entry Mode Set
0	0	0	0	0	0	1	D	C	B	Display On/Off
0	0	0	0	0	1	S/C	R/L	*	*	Cursor and Display shift
0	0	0	0	1	DL	N	F	*	*	Function Set
0	0	0	1	CG Address						Set CG RAM Address
0	0	1	DD Address						Set DD RAM Address	
0	1	BF	Address Counter						Read Busy Flag and Address	
1	0	Write Data						Write Data to CG or DD RAM		
1	1	Read Data						Read Data from CG or DD RAM		

**Tabelle 9.1.** Befehlssatz des HD44780 (\* Don't Care).

An das Display können entweder Befehle zur Konfiguration oder Zeichen zur Darstellung gesendet werden. Über das Signal *RS* wird festgelegt, ob das Kommando- oder das Datenregister angesprochen wird. Es ist zudem möglich, selbstdefinierte Zeichen in einem speziellen Speicher des HD44780 abzulegen.<sup>5</sup>

<sup>4</sup> Vgl. Datenblatt unter `<embedded-linux-dir>/datasheets/`.

<sup>5</sup> Dies geschieht mithilfe des *Character Graphics Ram (CG-RAM)*.

Von dieser Möglichkeit soll hier jedoch kein Gebrauch gemacht werden. Stattdessen wird als Zielspeicher nur das *Display Data Ram (DD-RAM)* beschrieben, dessen Inhalt direkt auf dem Display abgebildet wird. Ob die eingehenden Daten in das CG-RAM oder in das DD-RAM fließen, hängt davon ab, welcher Befehl zuletzt ausgeführt wurde: *Setze CG-RAM* oder *Setze DD-RAM*.

Der Adresszähler wird nach der Übertragung eines Zeichens vom HD44780 automatisch inkrementiert, sodass hintereinander geschriebene Zeichen auf dem Display nebeneinander stehen. Die Cursorposition kann aber auch bspw. für einen Zeilenwechsel direkt gesetzt werden. Weiterhin ist es möglich, Zeichen vom Display oder Inhalte der Zeichenmaske aus dem CG-RAM zurückzulesen. Diese Funktionalität wird aber eher selten benötigt. Tabelle 9.1 gibt eine Übersicht über den Befehlssatz des HD44780, Tabelle 9.2 liefert die Bedeutung der Parameter.

Bit	0	1
I/D	Cursorposition dekrementieren	Cursorposition inkrementieren
S	Anzeige steht still	Anzeige mitbewegen
D	Anzeige aus	Anzeige an
B	Cursor blinkt nicht	Cursor blinkt
C	Cursor aus	Cursor an
S/C	Cursor in Richtung R/L bewegen	Anzeige in Richtung R/L bewegen
R/L	Bewegung nach links	Bewegung nach rechts
DL	4-Bit-Modus	8-Bit-Modus
N	Anzeige einzeilig	Anzeige zweizeilig
F	5×8-Dot-Matrix	5×10-Dot-Matrix
BF	Display bereit	Display beschäftigt

**Tabelle 9.2.** Bedeutung der Parameter.

Wenn das Display gerade einen Befehl verarbeitet und noch nicht bereit für ein weiteres Kommando ist, so ist das Busy-Flag *BF* gesetzt. Eine Überprüfung des Flags ist sinnvoll, um die Ansteuerungsdauer so kurz wie möglich zu halten. Alternativ kann zwischen zwei Zugriffen eine kurze Zeit pausiert werden, bevor der nächste Befehl abgesetzt wird. Eine gewisse Latenz wird bereits durch die Übertragungszeiten auf dem I<sup>2</sup>C-Bus erreicht, sodass die Wartezeiten eher bei direktem Anschluss an einen Mikrocontroller relevant werden. Bei der Befehlsabfolge für die Initialisierung gilt dies allerdings nicht – hier sind generell längere Wartezeiten einzuhalten. Eine Abfrage des Busy-Flags ist zu diesem Zeitpunkt noch nicht möglich. Im 4-Bit-Modus werden Kommandos und Zeichen in die zwei Hälften eines Bytes, das High- und das Low-Nibble, zerlegt und nacheinander übertragen. Über eine steigende Flanke an der *Enable*-Leitung wird dem Display mitgeteilt, wann die Daten auf D4–D7 korrekt anliegen und übernommen werden sollen.

Für eine Initialisierung des Displays im 4-Bit-Modus müssen folgende Schritte durchgeführt werden:

1. Nach dem Einschalten 15 ms warten
2. Schreibe 0x30 an das Kommandoregister
3. Warte 5 ms
4. Schreibe nochmals 0x30 an das Kommandoregister
5. Warte weitere 100 µs
6. Schreibe nochmals 0x30 an das Kommandoregister
7. Schreibe 0x20 (4-Bit-Modus, zweizeilig, 5×8 Punkte)

Das dreimalige Schreiben von 0x30 führt einen internen Reset durch. Erst nach diesem Reset kann der Modus gewechselt werden. Wie bereits angedeutet, kann das Busy-Flag in der Initialisierungsphase noch nicht abgefragt werden. Aus diesem Grund sind die Pausen unbedingt einzuhalten.

### 9.3.6 Die Klasse IICDisplay

Die Klasse `IICDisplay` implementiert als Unterklasse von `IICBase` Zugriffsmethoden für ein LC-Display, welches über einen I<sup>2</sup>C-I/O-Expander vom Typ PCF8574 angeschlossen ist. Bei dem hier verwendeten Interface<sup>6</sup> ist das Display gemäß Tabelle 9.3 mit dem PCF8574 verbunden.

<b>PCF8574</b>	PCF 7	PCF 6	PCF 5	PCF 4	PCF 3	PCF 2	PCF 1	PCF 0
<b>Stiftleiste</b>	D7	D6	D5	D4	LCD Light	Enable	RW	RS

**Tabelle 9.3.** Schnittstellenbelegung zwischen PCF8574 und LC-Display bei Verwendung des PCF-LCD-Interface von CC-Tools.

Die Datenpins D0–D3 werden im 4-Bit-Modus nicht benötigt und bleiben frei. Platinen anderer Hersteller haben u. U. eine abweichende Belegung (vgl. bspw. das LCD-I<sup>2</sup>C-Interface von Conrad). Die Zuordnung wird daher in `IICDisplay.h` für den jeweiligen Fall definiert:

```
// define connection PCF8574 <-> LCD here
#define LCD_BACKLIGHT 0x08
#define LCD_ENABLE    0x04
#define LED_RW        0x02
#define LED_RS        0x01
#define LED_DATA_SHIFT 4
```

<sup>6</sup> PCF-LCD-Interface von CC-Tools, vgl. Komponentenliste in Tabelle E.2.

Die einzige Bedingung für die Anwendbarkeit der Definitionen ist eine zusammenhängende und aufsteigende Belegung der Datenpins D4–D7. Über das Makro `LED_DATA_SHIFT` wird die Anzahl der Bits angegeben, um welche sich der Beginn der Datenleitung D4 des Displays gegenüber PCF0 verschiebt. Alle anderen Makros enthalten die Wertigkeit der zugehörigen Pins des PCF8574.

Je nach Typ des Displays unterscheiden sich Größe und Adressierung des Displayspeichers. Ein Aufzählungstyp definiert mögliche Dimensionen der Zeichenmatrix:

```
enum iic_disp_type {
    SIZE_2x16,
    SIZE_4x16,
    SIZE_2x20,
    SIZE_4x20 };
```

Der passende Wert muss dem Konstruktor zusätzlich zu den Basisparametern übergeben werden. Für die Ablage des Displaytyps, des Beleuchtungs- und Anzeigestatus' sowie für die Bereitstellung eines Schreibpuffers für Anwendung und I<sup>2</sup>C-Schnittstelle sind für die Klasse `IICDisplay` folgende Membervariablen notwendig:

```
iic_disp_type m_disptype;           // display type (dimensions)
char m_backlight;                   // backlight status
char m_disp_control;                // display status
char m_disp_buf[DISPBUFLLENGTH];    // display character buffer
char m_buf;                         // i2c-rw-buffer
struct timespec m_ts_freeze_end;     // end of freeze time
// (no clear(), putChar() and setDisplay() operations while freeze)
```

In der Variablen `m_ts_freeze_end` wird ein Zeitpunkt angegeben, nach welchem der Displayinhalt frühestens wieder geändert werden darf. Bis zu diesem Zeitpunkt bleibt das Display „eingefroren“. Die für den Anwender relevanten Zugriffsmethoden sind in der folgenden API aufgelistet. Selten benötigte Displayfunktionen wurden nicht implementiert – die Klasse ist bei Bedarf um diese zu erweitern (ein Beispiel ist die Verschieberichtung in *Entry Mode Set*).

```
IICDisplay(IICBus& bus, int addr, string name, iic_disp_type
dtype);
```

Erzeugt ein Objekt vom Typ `IICDisplay` unter Angabe einer Referenz auf das Bus-Objekt `bus`, der I<sup>2</sup>C-Adresse `addr`, einer Bezeichnung `name` und des Displaytyps `dtype`.

```
void clear();
```

Löscht den Display-Inhalt und setzt den Cursor auf die Home-Position zurück.

```
void putChar(char ch);
```

Schreibt ein einzelnes Zeichen `ch` an die aktuelle Cursorposition.

```
void setCursorPos(int row, int col);
```

Setzt den Cursor in Zeile `row` und Spalte `col`, wobei die obere linke Ecke durch (0,0) repräsentiert wird.

```
void setCursorMode(bool curs, bool blink);
```

Stellt den Cursor an bzw. aus (`curs`) oder versetzt ihn in den Blinkmodus (`blink`).

```
void setDisplay(bool disp);
```

Aktiviert oder deaktiviert die Anzeige des Displays, die Speicherinhalte bleiben erhalten. Mit dem Abschalten während eines Aufrufs von `putChar()` lässt sich die schrittweise Darstellung einzelner Zeichen vermeiden.

```
void setBacklight(bool bl);
```

Stellt die Hintergrundbeleuchtung an oder aus.

```
char* getBuf();
```

Über diese Funktion wird ein Zeiger angefordert, um Daten im Displaypuffer abzulegen. Zu beachten ist, dass beim Schreiben in den Puffer der zulässige Bereich von `DISPBUFLength` nicht überschrieben wird.

```
void printBuf();
```

Schreibt alle im Puffer `m_disp_buf` enthaltenen Zeichen an die aktuelle Cursorposition. Ein Zeilenendezeichen `\n` bricht die Zeichenkette an dieser Stelle ab.

```
void freeze(unsigned int time_ms);
```

Friert das Display für eine bestimmte Zeit `time_ms` in Millisekunden ein, indem die Befehle `clear()`, `putChar()` und `setDisplay()` für diese Dauer ignoriert werden. Eine mögliche Anwendung ist bspw. die Ausgabe von Warnmeldungen während einer fortlaufenden Messwertausgabe. Eine Warnmeldung kann hiermit höher priorisiert werden und wird nicht sofort wieder überschrieben.

Auf die Implementierung der einzelnen Ansteuerungsroutinen soll hier nicht näher eingegangen werden. Der folgende Quelltext zeigt aber Grundlagen der Kommunikation mit dem Display. Das Senden eines Kommandos oder Zeichens erfolgt hierbei immer in zwei Schritten (vgl. auch die Methode `putChar()`, aus welcher das Snippet entnommen ist):

```
m_buf = (ch & 0xF0) >> (4 - LED_DATA_SHIFT); // HB: four upper bits
m_buf |= LED_RS; // set write
m_buf |= m_backlight;
writeData(&m_buf, 1);
trigger_enable();

m_buf = (ch & 0x0F) << LED_DATA_SHIFT; // LB: four lower bits
m_buf |= LED_RS; // set write
m_buf |= m_backlight;
writeData(&m_buf, 1);
trigger_enable();
```

Zuerst wird das *High-Nibble*<sup>7</sup> des zu sendenden Display-Daten-Bytes geschrieben. Dafür sind die untersten vier Bit zu löschen und die restlichen vier Datenbits an die richtige Position zu verschieben. In diesem Fall liegen sie mit `LED_DATA_SHIFT = 4` bereits an der richtigen Stelle. Nachdem `m.buf` die zu sendenden Display-Daten enthält, werden nun zusätzliche Steuerbits wie bspw. `LED_RS` gesetzt. Weiterhin wird der aktuelle Zustand der Hintergrundbeleuchtung addiert, da dieser durch den Schreibzugriff nicht verändert werden soll. Das vollständige PCF8574-Daten-Byte kann nun an den Ausgängen angelegt und durch Auslösen einer steigenden Flanke an `ENABLE` übernommen werden. Analog erfolgt in einem zweiten Schritt das Senden des *Low-Nibbles*. Der einzige Unterschied ist, dass die unteren vier Datenbits von D0 ausgehend in entgegengesetzter Richtung korrigiert werden müssen.

### 9.3.7 Die Klasse `IICIOExpander`

Nachdem nun die Klassen `IICKeyboard` und `IICDisplay` erklärt wurden, wird im folgenden Text eine weitere Klasse zur *generischen* Verwendung der PCF8574-Bausteine vorgestellt. Für eigene I/O-Entwicklungen auf Basis eines PCF8574 steht `IICIOExpander` mit folgender API zur Verfügung:

```
IICIOExpander(IICBus& bus, int addr, string name);
```

Erzeugt ein Objekt vom Typ `IICIOExpander` unter Angabe einer Referenz auf das Bus-Objekt `bus`, der I<sup>2</sup>C-Adresse `addr` und einer Bezeichnung `name`.

```
int setPort(char data);
```

Schreibt die in `data` angegebenen Daten in das I/O-Register.

```
int setPortOn(char pins);
```

Setzt die in `pins` markierten Ausgänge auf logisch 1.

```
int setPortOff(char pins);
```

Setzt die in `pins` markierten Ausgänge auf logisch 0.

```
int getPort();
```

Liefert im niederwertigen Byte den Zustand des kompletten I/O-Registers zurück.

```
int getPin(char pin);
```

Gibt den Zustand eines einzelnen Pins `pin` zurück (`pin` ist im Bereich 0..7 anzugeben).

Falls keine anderen Rückgabewerte spezifiziert wurden, so liefern die Funktionen die Anzahl der geschriebenen oder gelesenen Daten-Bytes zurück, im Fehlerfall `-1`. Das Beispiel `io_expander` zeigt die Verwendung dieser Klasse.

---

<sup>7</sup> Mit *Nibble* werden vier Bit bezeichnet, in diesem Fall die vier mit der höchsten Wertigkeit im Daten-Byte.



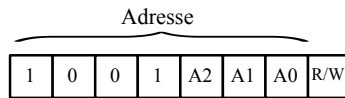
## 9.4 Temperaturmessung mit DS1631

### 9.4.1 Dallas DS1631

Der DS1631 von Dallas ist ein digitales Thermometer mit I<sup>2</sup>C-Schnittstelle und wird als Standardbaustein in Lüftersteuerungen oder auf PC-Mainboards eingesetzt. Der Baustein verfügt über eine Thermostatfunktion mit separatem Schaltausgang und arbeitet sofort nach Anlegen einer Versorgungsspannung im *Stand-Alone*-Betrieb. Folgende Eigenschaften zeichnen den DS1631 aus:

- Arbeitsbereich:  $-55^{\circ}\text{C}$  bis  $+125^{\circ}\text{C}$  bei  $\pm 1^{\circ}\text{C}$  Genauigkeit, bzw.  $0^{\circ}\text{C}$  bis  $+70^{\circ}\text{C}$  bei  $\pm 0,5^{\circ}\text{C}$  Genauigkeit
- Konfigurierbare Messgenauigkeit: 9, 10, 11 oder 12 bit
- Versorgungsspannung: 2,7 V bis 5,5 V
- Thermostatfunktion mit separatem Ausgang  $T_{OUT}$
- Wandlung in Einzelschritten oder kontinuierlich

Die 7-Bit-Adresse wird hardwareseitig über drei Pins festgelegt (vgl. Abbildung 9.5). Entsprechend können bis zu acht Sensoren dieses Typs an einem Bus betrieben werden.



**Abb. 9.5.** Adressregister des DS1631.

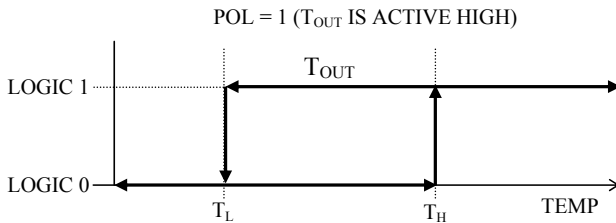
Nach dem Start ist die Auflösung des Bausteins automatisch auf 12 bit gesetzt, kann aber über das Konfigurationsregister auf bis zu 9 bit gedrosselt werden. Die Verringerung der Genauigkeit geht mit einer schnelleren Wandlung einher. Tabelle 9.4 veranschaulicht den Zusammenhang.

R1	R0	Auflösung	Wandlungszeit
0	0	9 bit	93,75 ms
0	1	10 bit	187,5 ms
1	0	11 bit	375 ms
1	1	12 bit	750 ms

**Tabelle 9.4.** Wandlungszeiten bei verschiedenen Auflösungen.

Nach der Wandlung wird der Temperaturwert als Zweierkomplement in den Registern  $T_H$  und  $T_L$  abgelegt, wobei der Hexadezimalwert 0x7D00 einer Temperatur von  $125^{\circ}\text{C}$  entspricht. Ein Betrieb mit einer Auflösung von 12 bit, also

theoretisch  $0,06^{\circ}\text{C}$ , ist eher für eine vergleichende Messung geeignet; die absolute Genauigkeit ist kleiner. Abbildung 9.6 zeigt die Thermostatfunktion des Ausgangs  $T_{\text{OUT}}$ . Der Anwender kann hierbei die Schwellwerte  $T_L$  und  $T_H$  sowie die Schaltlogik vorgeben. Diese Daten, wie auch der eingestellte Modus, werden im EEPROM gespeichert und sind beim nächsten Start sofort verfügbar. Beim Rücksetzen der Temperatur-Flags TLF und THF ist darauf zu achten, dass auch jedes Mal das EEPROM neu geschrieben wird. Zu viele Schreibvorgänge sind entsprechend zu vermeiden.



**Abb. 9.6.** Thermostatfunktion mit Hysterese, definiert durch  $T_L$  und  $T_H$ . Quelle: *Dallas Semiconductor*, Datenblatt DS1631.

Anzumerken ist, dass der Baustein im SMD-Gehäuse geliefert wird. Wer sich die SMD-Lötarbeiten ersparen möchte, kann auch mit dem C-Control-Modul von Conrad eine fertig bestückte DS1631-Platine mit Schraubanschlüssen für Versorgung und I<sup>2</sup>C-Datenleitungen beziehen (vgl. Tabelle E.2).

#### 9.4.2 Die Klasse `IICTempSensor`

Die Klasse `IICTempSensor` repräsentiert Objekte vom Typ eines DS1631-Temperatursensors. Die Klasse lässt sich relativ einfach auch für ähnliche Sensortypen wie bspw. den LM75 anpassen. Um die Werte des Konfigurationsregisters sowie Informationen über die eingestellte Auflösung und die Polarität des Thermostat-Ausgangs vorzuhalten, werden folgende Objektvariablen benötigt:

```
char m_conf_reg; // configuration register
char m_res;      // resolution as 2-bit-value
bool m_pol;      // thermostat output polarity
char m_buf[3];   // rw-buffer
```

Beim Anlegen eines Objektes vom Typ `IICTempSensor` sind neben den Standardparametern die gewünschte Messauflösung `res` als 2-Bit-Wert gemäß Tabelle 9.4 sowie die Polarität für den Thermostat-Ausgang `t_out_pol` zu übergeben.

Die DS1631-Bausteine werden in dieser Klasse immer im *Continuous Mode* betrieben. Als API stehen die folgenden Methoden zur Verfügung:

```
IICTempSensor(IICBus& bus, int addr, string name);
```

Erzeugt ein Objekt vom Typ `IICTempSensor` unter Angabe einer Referenz auf das Bus-Objekt `bus`, der I<sup>2</sup>C-Adresse `addr` und einer Bezeichnung `name`.

```
int startConvert();
```

Beginnt mit einer kontinuierlichen Umwandlung.

```
int stopConvert();
```

Stoppt eine kontinuierliche Umwandlung.

```
int setThermostat(int t_low, int t_high);
```

Setzt die Thermostatschwellen auf die in `t_low` und `t_high` angegebenen Temperaturwerte. Diese Methode bewirkt einen EEPROM-Zugriff und sollte deshalb nicht zyklisch ausgeführt werden.

```
int resetTempFlags();
```

Setzt die Flags TLF und THF zurück.

```
int readTemp();
```

Liefert im Erfolgsfall die zuletzt konvertierte Temperatur als Ganzzahl in ( $^{\circ}\text{C} \cdot 10$ ) zurück, sonst  $-999$ .

```
bool getTempFlag(bool highlow);
```

Liefert das Temperatur-Flag der unteren Grenze TLF (`highlow=0`), bzw. der oberen Grenze THF (`highlow=1`), zurück.

Falls nicht anders definiert, wird im Erfolgsfall die Anzahl kommunizierter Daten-Bytes zurückgeliefert, sonst  $-1$ .

Die Temperaturen werden als Ganzzahlen in der Einheit ( $^{\circ}\text{C} \cdot 10$ ) übertragen. Damit erübrigt sich die Fließkommadarstellung und es ergibt sich auf manchen Embedded-Systemen ein gewisser Geschwindigkeitsvorteil. Das Beispiel `temp_sensor` zeigt das zyklische Auslesen eines DS1631-Sensors.

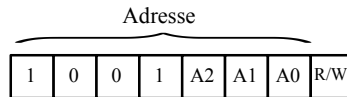
## 9.5 A/D- und D/A-Wandler

### 9.5.1 Philips PCF8591

Beim Philips PCF8591 handelt es sich um einen Wandlerbaustein mit vier analogen Eingängen, einem analogen Ausgang und einer Auflösung von jeweils 8 bit. Die Eingänge können wahlweise als einfache Eingänge mit Massebezug, als differentielle Eingänge mit gemeinsamer externer oder eigener Referenz oder im Mischbetrieb verwendet werden. Für den Analogteil ist eine stabile Versorgungsspannung an  $V_{ref}$  und  $A_{GND}$  zu legen. Werden die Eingänge im differentiellen Modus betrieben, dann sind nur drei (bei gemeinsamer Referenz) bzw. zwei (bei eigener Referenz) Eingänge verfügbar. Die Konfiguration wird

zusammen mit Angaben über die A/D-Kanalnummer, dem Autoincrement-Modus und einem Aktivitätsbit für den Ausgang in einem Kontroll-Byte übertragen.

Im Autoincrement-Modus wird nach jedem gesendeten Daten-Byte automatisch der A/D-Kanal erhöht. Wird der A/D-Ausgang nicht benötigt, kann dieser über das entsprechende Bit abgeschaltet werden. Dies bewirkt eine Absenkung des Stromverbrauchs auf unter 1 mA. Abbildung 9.7 zeigt die Basisadresse des PCF8591 mit drei konfigurierbaren Bits.



**Abb. 9.7.** Adressregister des PCF8591.

Die A/D-Wandlung wird automatisch nach der Übertragung einer gültigen Leseanforderung gestartet. Während des Sendens des zuletzt konvertierten Daten-Bytes wird das nachfolgende Byte gewandelt. Dieser Umstand des „veralteten“ ersten Daten-Bytes sollte bei zeitkritischen Messungen beachtet werden.

### 9.5.2 Die Klasse IICADConverter

Die Klasse `IICADConverter` repräsentiert I<sup>2</sup>C-Bausteine vom Typ PCF8591 und stellt entsprechende Erweiterungen der Basisklasse zur Verfügung. In der Klassenbeschreibung `ADConverter.h` werden die Modi der analogen Betriebsarten über einen Aufzählungstyp definiert:

```
enum eADConverterMode {
    SINGLE_ENDED_INPUT,           // four single ended inputs
    THREE_DIFF_INPUTS,           // three differential inputs
    SINGLE_DIFF_MIXED,           // single ended and differential mixed
    TWO_DIFF_INPUTS              // two differential inputs
};
```

In den folgenden Membervariablen werden das Kontrollregister, der aktuell ausgewählte A/D-Kanal, der verwendete Modus und die Autoincrement-Option abgelegt:

```
char m_control_byte;           // control byte
int m_channel;                 // current AD-channel
int m_mode;                   // AD input mode
bool m_autoinc;               // autoincrement flag
map<int,int> scoreTable[MAXCH]; // score table for sensor charact.
```

Weiterhin sind mehrere Wertetabellen vom Typ `map<int,int>` definiert, welche der Hinterlegung von Kennlinien für A/D-Kanäle bzw. dort angeschlossene Sensoren dienen. Die Tabellen müssen zu Beginn der Anwendung über `setScoreTable()` mit Wertepaaren gefüllt werden. Eine Abfrage ist mittels `getScoreTable()` möglich. Zwischen zwei hinterlegten Messpunkten erfolgt eine lineare Interpolation.

Um Berechnungen möglichst effizient zu gestalten, werden in `scoreTable` nur Ganzzahlen abgelegt. Bei der Repräsentation physikalischer Größen müssen die Einheiten entsprechend skaliert werden. Für den Anwender stellt die Klasse `IICADConverter` eine API mit folgenden Methoden zur Verfügung:

```
IICADConverter(IICBus& bus, int addr, string name,
               eADConverterMode mode);
```

Neben der üblichen Angabe des Bus-Objekts, einer I<sup>2</sup>C-Adresse und der Bezeichnung muss in `mode` der Modus für die A/D-Wandlung festgelegt werden. Das Autoincrement-Flag ist zu Beginn standardmäßig abgeschaltet.

```
int setAutoInc(bool autoinc);
```

Setzt das Autoincrement-Flag und überträgt anschließend das Kontroll-Byte. Ein Rückgabewert von  $-1$  signalisiert ein Lese- oder Schreibproblem.

```
int readChannel(int channel, bool read_immediate, int
               table_num);
```

Gibt im manuellen Kanalmodus den letzten konvertierten Wert ( $0 - 255$ ) für Kanal `channel` zurück. Um einen aktuellen Wert zu erhalten, ist `read_immediate` auf `true` zu setzen. Hiermit werden zwei Bytes gelesen und der neuere Wert zurückgeliefert. Im Autoincrement-Modus ist die Angabe von `channel` prinzipiell unbedeutend. Sinnvoll ist aber eine Übergabe von  $-1$ , um beim späteren Umschalten in den manuellen Modus einen Kanalwechsel zu erzwingen. Über `table_num` kann angegeben werden, ob der Registerwert ( $-1$ ) oder ein mittels `getScoreTable()` umgerechneter Wert ( $\geq 0$ ) zurückgegeben wird. Schlägt der Vorgang fehl, so wird  $-1$  zurückgeliefert.

```
int setChannel(int channel);
```

Setzt den aktiven A/D-Kanal auf `channel`, falls dieser nicht bereits ausgewählt ist. Werte ausserhalb des verfügbaren Bereiches (man beachte die unterschiedlichen A/D-Modi) werden durch den Rückgabewert  $-1$  quittiert. Im Erfolgsfall wird die neue Kanalnummer zurückgegeben.

```
int setOutput(int value);
```

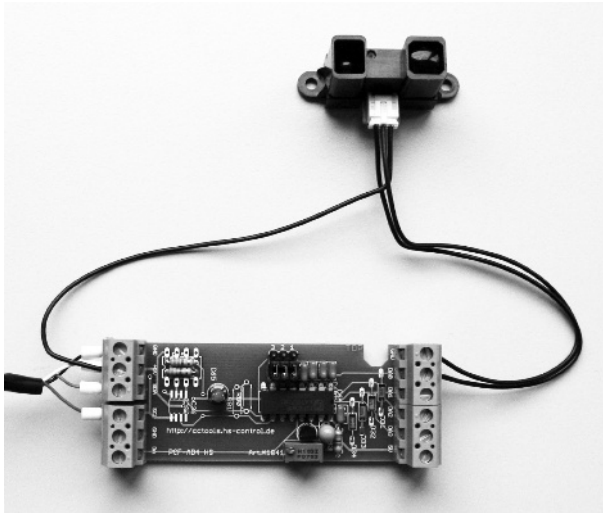
Setzt den A/D-Ausgang auf den Registerwert `value`. Ist der Ausgang abgeschaltet, wird dieser automatisch aktiviert. Bei einem Übergabewert  $-1$  wird der Ausgang deaktiviert. Ein Rückgabewert  $-1$  signalisiert ein Lese- oder Schreibproblem. Im Erfolgsfall wird die Anzahl übertragener I<sup>2</sup>C-Bytes zurückgeliefert.

```
void setScoreTable(int v1, int v2, int channel);
```

Fügt der Tabelle `scoreTable[channel]` ein Wertepaar `v1,v2` hinzu. Bei `v1` handelt es sich um den Registerwert eines A/D-Kanals, bei `v2` um den Funktionswert an dieser Stelle. Das Füllen der Wertetabellen sollte zu Beginn einer Anwendung im zeitlich unkritischen Teil erfolgen.

```
int getScoreTable(int value, int channel);
```

Liefert zu einem Messwert `value` für einen bestimmten Kanal `channel` den hinterlegten Wert aus Tabelle `scoreTable[channel]` zurück. Zwischen zwei Tabelleneinträgen wird linear interpoliert. Sollte der Messwert außerhalb des Bereichs liegen, wird dieser auf die Grenzen abgebildet. Diese Funktion wird von `readChannel()` verwendet und muss entsprechend bei regulärer Anwendung nicht aufgerufen werden. Durch die Integration in die API ist jedoch auch eine Verwendung der Umrechnungstabellen für andere Zwecke möglich.



**Abb. 9.8.** PCF-AD4-HS von CC-Tools mit Sharp-Distanzsensor GP2Y0A02 [CC-Tools 07].

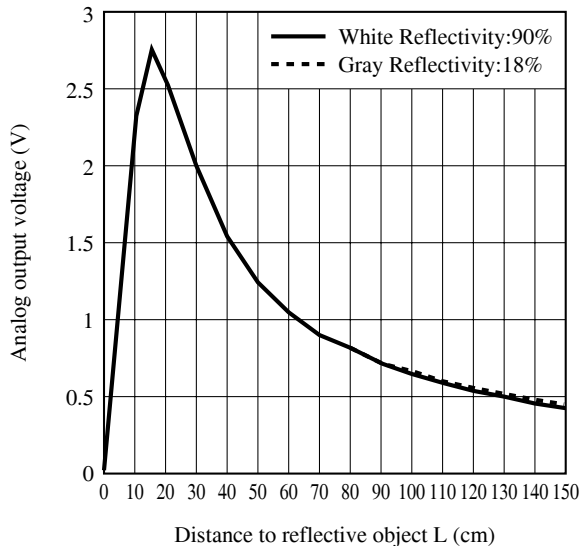
Im Beispiel `distance_measuring` wird ein I<sup>2</sup>C-Baustein PCF8591 verwendet, um die Distanzinformation einer Infrarotlichtschranke vom Typ Sharp GP2Y0A02 auszulesen. Abbildung 9.8 zeigt die PCF-AD4-HS-Platine von CC-Tools mit angeschlossener Lichtschranke (vgl. auch die Tabellen E.1 und E.2 im Anhang).

Der Sensor wird mit 5 V versorgt und liefert eine Ausgangsspannung zwischen 0 und 2,8 V bei einem Messbereich von 20–150 cm. Abbildung 9.9 zeigt den Zusammenhang zwischen Ausgangsspannung und Distanz des Sensors. Die zu-

gehörige Kennlinie wird in Form einer Wertetabelle hinterlegt. In dieser Tabelle werden zu Beginn Punktpaare als Einträge in folgender Form für Kanal 0 hinzugefügt:

```
converter.setScoreTable(<messwert>, <ausgabewert>, 0);
...
```

Als `<messwert>` dient der Registerwert. Die Spannungswerte in Abbildung 9.9 werden deshalb zu `<messwert>=  $\frac{V_{out}}{V_{ref}} \cdot 255$`  berechnet. Da die Ausgabe in der Einheit Zentimeter erfolgen soll, kann `<ausgabewert>` direkt auf den zugehörigen Abstand gesetzt werden.



**Abb. 9.9.** Zusammenhang zwischen Ausgangsspannung und gemessener Distanz für den Infrarot-Abstandssensor Sharp GP2Y0A02. Quelle: Sharp, Datenblatt GP2Y0A02.

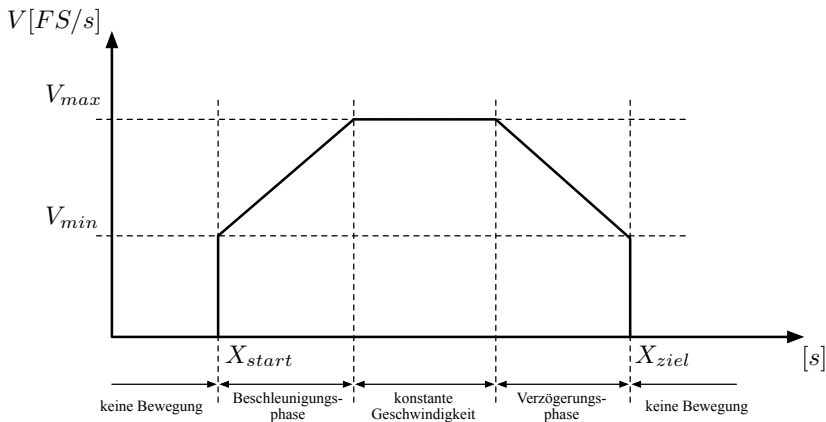
Aufgrund der Doppeldeutigkeit am Anfang der Kennlinie wird nur der Messbereich oberhalb 20 cm in der Wertetabelle hinterlegt (vgl. Abbildung 9.9). Im Beispiel lässt sich mit einer Abtastung von 1 Hz die Latenz einer Abfrage mit `readChannel(x,0,x)` und `readChannel(x,1,x)` sehr gut nachvollziehen.

## 9.6 TMC222-Schrittmotorsteuerung

### 9.6.1 Trinamic TMC222

Bei einem Schrittmotor werden die Spulenwicklungen in einer bestimmten Abfolge derart angesteuert, dass sich ein Drehfeld ergibt. Dieses wiederum erzeugt die Drehbewegung des Ankers. Die Aufgabe der Spulenansteuerung wird oft von Mikrocontrollern übernommen und erfordert im Zusammenspiel mit einem Treiber-IC<sup>8</sup> in der Regel eine Vorgabe der Schrittzahl und der Drehrichtung. Die positionsgenaue und gleichförmige Erzeugung einer bestimmten Bewegung kann hierbei einen Mikrocontroller fast völlig auslasten.

Ein Controller, der speziell für diese Aufgabe entworfen wurde, ist der TMC222 der Firma Trinamic [Trinamic 08]. Der Baustein bietet neben der Motortreiber-Endstufe für bipolare Schrittmotoren (vier Anschlüsse) auch einen sog. *Motion Controller*. Dieser Logikteil koordiniert den Bewegungsablauf und damit alle zeitkritischen Aufgaben. Der TMC222 stellt dem steuernden Rechner eine abstrakte Schnittstelle in Form von Zielposition, Beschleunigung und Maximalgeschwindigkeit zur Verfügung (vgl. Abbildung 9.10).



**Abb. 9.10.** Rampenerzeugung des Trinamic TMC222. Quelle: Trinamic, Datenblatt TMC222.

Über integrierte RAM- und OTP<sup>9</sup>-Speicher können Bewegungsparameter abgelegt werden. Die Kapselung dieser Low-Level-Befehle entlastet den Host-

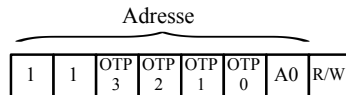
<sup>8</sup> Bspw. dem Schrittmotortreiber A3979 von Allegro, vgl. <http://www.allegromicro.com>.

<sup>9</sup> Engl. *One Time Programmable*.



Controller und ermöglicht zudem die Ansteuerung über ein integriertes serielles I<sup>2</sup>C-Interface.<sup>10</sup> Folgende Eckdaten kennzeichnen den TMC222:

- Betrieb von bipolaren Schrittmotoren im Micro-Step-Modus
- Spulenspannung bis 800 mA
- Versorgungsspannung 8–29 V
- Diagnose für Übertemperatur, Leitungsbruch, Kurzschluss, Überstrom und Unterspannung
- Motion-Controller mit internem 16-Bit-Zähler
- Rampengenerator, Vorgabe der Beschleunigung und der Geschwindigkeit
- Anschluss für Referenzschalter
- I<sup>2</sup>C-Schnittstelle mit bis zu 350 kbit/s



**Abb. 9.11.** I<sup>2</sup>C-Adressfeld des TMC222.

Abbildung 9.11 zeigt das Adressfeld des TMC222. Über das OTP-Register lassen sich vier Bit fest programmieren, die bei der Auslieferung auf 0 gesetzt sind. Bit 0 kann in der Schaltung verdrahtet werden und wird in der Regel auf einen Jumper herausgeführt. Werden mehrere TMCs an einem I<sup>2</sup>C-Bus betrieben, sind die OTP-Felder zuvor entsprechend zu parametrieren. Insgesamt können dann maximal 32 TMC222-Module verwendet werden. Zur Kommunikation mit dem Logikteil des TMC222 stehen dem Anwender die in Tabelle 9.5 aufgelisteten Befehle zur Verfügung.

### 9.6.2 Conrad C-Control I<sup>2</sup>C-Bus-Stepper-Driver

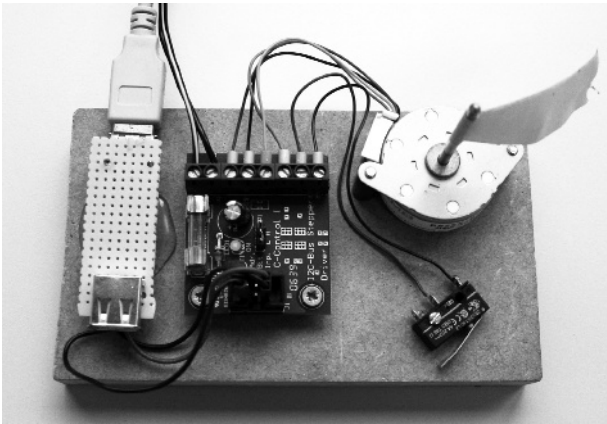
Mit dem Conrad I<sup>2</sup>C-Bus-Stepper-Driver ist ein günstiges Applikationsboard auf Basis des TMC222 erhältlich (vgl. Tabelle E.2). Der Anwender erspart sich hiermit das Auflöten des SMD-Bausteins oder gar ein eigenes Platinenlayout. Zudem sind auf der Stepper-Driver-Platine zusätzliche Komponenten zur Spannungsstabilisierung und Absicherung untergebracht. Bei Motoren bestimmter Bauform ist es möglich, das Modul direkt am Gehäuse zu befesti-

<sup>10</sup> Unter der Bezeichnung TMC211 ist der Baustein auch mit LIN-Bus-Anschluss verfügbar.

Befehl	Funktion	Befehls-Byte
GetFullStatus1	Liefert den kompletten Status zurück	0x81
GetFullStatus2	Liefert Ist-, Soll- und Sicherheitsposition	0xFC
GetOTPParam	Liefert OTP-Parameter	0x82
GotoSecurePosition	Führt den Motor an die Sicherheitsposition	0x84
HardStop	Stoppt den Motor sofort	0x85
ResetPosition	Setzt die aktuelle Ist- und Zielposition auf 0	0x86
ResetToDefault	Überschreibt RAM-Speicher mit OTP-Inhalt	0x87
RunInit	Referenzfahrt	0x88
SetMotorParam	Setzt die Motorparameter	0x89
SetOTPParam	Löscht den OTP-Speicher	0x90
SetPosition	Setzt Ziel- und Sicherheitsposition	0x8B
SoftStop	Stoppt den Motor mit eingestellter Verzögerung	0x8F

**Tabelle 9.5.** Befehlssatz des TMC222.

gen.<sup>11</sup> Abbildung 9.12 zeigt einen Versuchsaufbau mit Motor, Stepper-Driver und Endschalter.



**Abb. 9.12.** I<sup>2</sup>C-Bus-Stepper-Driver mit Schrittmotor und Endschalter.

Der I<sup>2</sup>C-Bus wird gemäß Tabelle 9.6 über einen Pfostenstecker angeschlossen. Versorgung, Motoranschlüsse und Endschaltereingang sind auf Schraubklemmen herausgeführt. Die Leitung für den Endschalter kann wahlweise auf GND oder V+ gesetzt werden – eine interne Logik bestimmt automatisch den richtigen Schaltzustand. Falls die Belegung des verwendeten Motors nicht bekannt ist, können die Spulenanschlüsse durch Widerstandsmessung mithilfe eines Multimeters bestimmt werden. Die korrekte Polung kann im Betrieb er-

<sup>11</sup> Dies gilt für Motoren mit einem Abstand von 31 mm zwischen den Bohrungen.

mittelt werden: Bei einer ungleichmäßigen Bewegung ist wahrscheinlich eine der beiden Spulen nicht richtig gepolt.

Anschluss	Beschreibung
SDA (PF), Pin 5	I <sup>2</sup> C-Bus Datenleitung
SCL (PF), Pin 6	I <sup>2</sup> C-Bus Taktleitung
GND (PF), Pin 1	I <sup>2</sup> C-Bus Signalmasse
V+	Versorgungsspannung, 8 V–24 V
GND	Versorgungsmasse
OB2	Spule B, negativer Anschluss
OB1	Spule B, positiver Anschluss
OA2	Spule A, negativer Anschluss
OA1	Spule A, positiver Anschluss
ERS	Endschalter

**Tabelle 9.6.** Anschlussbelegung des I<sup>2</sup>C-Bus-Stepper-Moduls von Conrad. Die I<sup>2</sup>C-Leitungen SDA, SCL und Signalmasse werden auf dem sechspoligen Pfostenstecker (PF) aufgesteckt. Für die anderen Anschlüsse existieren Schraubklemmen.

### 9.6.3 Die Klasse IICStepper

Mit der Klasse `IICStepper` wird dem Anwender eine einfache Schnittstelle geboten, um Schrittmotoren über einen TMC222-Baustein anzusteuern. Die Anwendung eignet sich ebenfalls für andere Layouts und ist nicht auf das Applikationsboard von Conrad beschränkt. Um die Implementierung auch für eigene Erweiterungen nachvollziehbar zu gestalten, werden die in Tabelle 9.5 aufgelisteten TMC222-Befehle als gleichnamige Methoden implementiert. Eine Ausnahme bilden die beiden OTP-Zugriffsfunktionen `getOTPPParam()` und `setOTPPParam()`. Diese werden zum Schreiben des OTP-Speichers in einer separaten Funktion verwendet (vgl. auch Abschnitt 9.6.4).

Für die einfache Verwendung des Bausteins setzen zusätzliche Methoden auf den TMC222-Befehlen auf. So werden auch Status und Parameter des TMC222 für einen einfachen Zugriff in separaten Strukturen `motor_params` und `motor_status` hinterlegt.

Nach einer kombinierten Abfrage mittels `getFullStatus2()` für den direkten Zugriff auf die Einzelpositionen (Soll-, Ist- und Sicherheitsposition) werden auch diese als Membervariablen gespeichert. Mit einem umfangreichen I<sup>2</sup>C-Puffer `m_buf` von zehn Bytes ergeben sich für die Klasse `IICStepper` folgende Objektvariablen:

```
motor_params params;           // motor parameters
motor_status status;          // motor status
short actual_pos, target_pos, secure_pos; // 16-bit-positions
char buf[10];                 // rw-buffer
```

Als Benutzerschnittstelle sind folgende Methoden implementiert:

`IICStepper(IICBus& bus, int addr, string name);`

Erzeugt ein Objekt vom Typ `IICStepper` unter Angabe der Standardparameter für den zugeordneten I<sup>2</sup>C-Bus `bus`, der Bausteinadresse `addr` und der Bezeichnung `name`.

#### TMC222-Befehle

Bis auf die Befehle `getOTPParam()` und `setOTPParam()` stehen alle in Tabelle 9.5 aufgelisteten Befehle auch als Memberfunktionen zur Verfügung. Diese sind hier nicht nochmals einzeln aufgeführt.

`motor_status getStatus();`

Liefert die Statuswerte in einer Struktur vom Typ `motor_status` gesammelt zurück. Die Daten werden direkt vom TMC222 ermittelt.

`motor_params getParams();`

Liefert die Parameter in einer Struktur vom Typ `motor_params` gesammelt zurück. Die Daten werden direkt vom TMC222 ermittelt.

**Wichtig:** Für die folgenden Zugriffsfunktionen müssen die Daten zunächst mittels `getFullStatus1()` bzw. `getFullStatus2()` vom TMC222 gelesen werden:

`unsigned char getTempInfo();`

Liefert den Temperaturstatus als 2-Bit-Wert zurück.

`bool getExtSwitch();`

Liefert den Zustand des Endschalters (gedrückt = 1).

`bool getMotion();`

Gibt an, ob der Motor sich momentan in Bewegung befindet.

`short getActualPos(), getTargetPos(), getSecurePos();`

Liefert die jeweilige Position als 16-Bit-Wert zurück.

`int referenceInit(motor_params params, unsigned char v_max, unsigned char v_min, short ref_dist, short null_dist);`

Führt eine Referenzfahrt zum Endschalter aus. Der Motor bewegt sich dabei um eine maximale Distanz von `ref_dist` Schritten. Wird der Endschalter auf dieser Strecke nicht erreicht, erfolgt ein Abbruch mit Rückgabewert `-1`. Bei Erreichen des Endschalters wird der TMC222 neu initialisiert und es erfolgt eine Fahrt zur relativen Position `null_dist`. Jetzt wird der Zähler zurückgesetzt. Die Referenzfahrten werden mit `v_min` und `v_max` ausgeführt. Anschließend werden die Parameter auf `params` gesetzt.

`int waitForStop(unsigned int maxtime_ms);`

Wartet blockierend bis die Bewegung beendet ist, allerdings höchstens `maxtime_ms` Millisekunden. Eine Auswertung erfolgt durch kontinuierliches

Lesen des Motion-Flags. Kommt der Motor innerhalb der Zeitvorgabe zum Stillstand, wird 0 zurückgegeben, sonst  $-1$ .

```
void printParam();
```

Gibt alle Parameter auf der Standardausgabe aus.

```
void printStatus();
```

Gibt alle Statuswerte auf der Standardausgabe aus.

Wenn die Rückgabewerte nicht näher spezifiziert wurden, wird im Erfolgsfall die kommunizierte Anzahl der I<sup>2</sup>C-Daten-Bytes zurückgegeben, sonst  $-1$ . Es ist zu beachten, dass der Aufruf der notwendigen TMC222-Befehle nicht automatisch in die `get`-Methoden für einzelne Variablenzugriffe integriert ist. Da teilweise bis zu acht Bytes über den Bus übertragen werden, sollte dies nicht häufiger als notwendig geschehen. Es liegt in der Hand des Anwenders, den Zeitpunkt zu wählen, wann Daten aktualisiert werden sollen. Vor dem Zugriff auf Parameter oder Statusvariablen ist ein Aufruf von `getFullStatus1()` notwendig. Eine Positionsabfrage benötigt einen vorherigen Aufruf von `getFullStatus2()`.

Nach dem Neustart eines TMC222 sind vor der ersten Motorbewegung folgende Initialisierungsschritte auszuführen:

1. Aufruf von `getFullStatus1()`; die Schaltung verlässt den Sicherheitszustand.
2. Abfrage der aktuellen Position durch einen Aufruf von `getFullStatus2()`; die Sollposition für `runInit()` muss sich davon unterscheiden.
3. Setzen der notwendigen Bewegungsparameter durch einen Aufruf von `setMotorParam()`.
4. Start der Initialisierungsphase durch eine Ausführung von `runInit()` bzw. `referenceInit()`, falls ein Endschalter angefahren werden soll.

Das Beispiel `stepper` zeigt, welche Schritte nach der Erstellung eines Objektes vom Typ `IICStepper` notwendig sind, um eine korrekte Referenzfahrt mit anschließender Positionierung auszuführen.

#### 9.6.4 Programmierung des TMC222-OTP-Speichers

Wie bereits in Abschnitt 9.6 angesprochen, können vier Bits der TMC-Adresse über den OTP-Speicher fest programmiert werden. Dies ist nur einmalig möglich und sollte deshalb nicht in der Anwendung selbst geschehen. Mit dem zweiten Quelltextbeispiel in `stepper` steht ein Programm nur für diese Aufgabe zur Verfügung (Datei `otptool.c`). Es empfiehlt sich, zunächst etwas Erfahrung zu den Parameterwerten für Maximalstrom, Beschleunigung

und Geschwindigkeit zu sammeln, bevor diese gemeinsam mit der Adresse fest eingebrannt werden.

Bei einem Neustart wird der RAM-Inhalt aus dem OTP-Speicher kopiert, womit sich mit fest eingestellten Variablen ein Aufruf von `setMotorParam()` zu Beginn erübrigt. Versehentlich fehlerhaft programmierte OTP-Werte sind zwar ärgerlich, aber nicht tragisch. Falls zumindest die Adresse korrekt ist, so können die Werte im RAM wieder überschrieben werden. Über den Befehl `resetToDefault()` kann der RAM-Inhalt auf OTP-Werte zurückgesetzt werden. Das C-Programm `otptool` lässt sich über ein im Makefile separat definiertes Ziel übersetzen:

```
$ make otptool
```

Im Listing sind die nachfolgenden Parameter bereits definiert. Die Werte sind für die eigene Applikation entsprechend anzupassen:

```
#define ADDRESS 0x60
#define OSC      0x08
#define IREF     0x04
#define TSD      0x00
#define BG       0x00
#define AD       0x00
#define IRUN     0x0B
#define IHOLD    0x03
#define VMAX     0x04
#define VMIN     0x01
#define SECPOS   500
#define SHAFT    0x01
#define ACC      0x02
#define STEPMODE 0x03
```

Der OTP-Inhalt besteht aus insgesamt acht Bytes.<sup>12</sup> Diese werden zunächst aus den definierten Makros zusammengesetzt und dann, nach erfolgreicher Detektion des TMC222-Bausteins, nacheinander geschrieben. Am Ende des Programms wird der OTP-Inhalt zurückgelesen und ausgegeben. Zunächst sollte ein Testlauf erfolgen um die zusammengesetzten Werte zu überprüfen. Anschließend kann folgende Einkommentierung vorgenommen und ein Brennversuch gestartet werden:

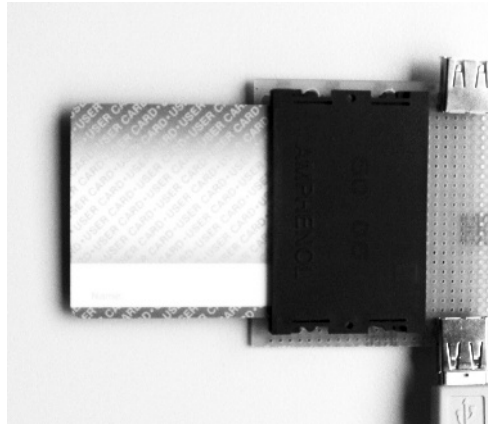
```
// #define ACTIVE
```

## 9.7 Chipkarten-Ansteuerung

EEPROM-Speicher in Chipkartenform werden auf Bankkarten, Krankenkassenkarten und in SIM-Karten von Mobiltelefonen verwendet. Mit Speichergrößen bis zu 64kByte lassen sich nicht allzu viele Informationen darauf abspeichern. Für persönliche Daten wie Adresse oder Identifikationsnummern

<sup>12</sup> Vgl. Tabelle 9 im Datenblatt `<embedded-linux-dir>/datasheets/tmc222.pdf`.

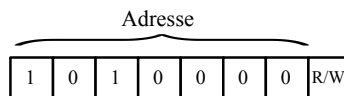
reicht dies jedoch aus. Für die Integration in eigene Anwendungen steht eine Vielzahl von Möglichkeiten offen. Beispiele sind die Umsetzung einer Zugangskontrolle oder eines Abrechnungssystems. Da EEPROM-Chipkarten über eine I<sup>2</sup>C-Schnittstelle angesprochen werden, bietet sich eine Eingliederung in das vorgestellte Buskonzept an. Abbildung 9.13 zeigt eine Chipkarte mit Kontaktiereinrichtung (vgl. hierzu auch die Bezugsquellentabelle in Anhang E).



**Abb. 9.13.** EEPROM-Chipkarte AT24C128SC mit Kontaktiereinrichtung.

### 9.7.1 EEPROM-Chipkarte AT24Cxx

Die EEPROM-Chipkarten der Reihe AT24Cxx von Atmel sind mit Speichergrößen zwischen 2 kByte und 64 kByte erhältlich und entsprechen dem Standard ISO 7816 für Smart Cards. In diesem Abschnitt wird eine Karte AT24C128SC mit 16 kByte Speicher (128 kbit) verwendet. Die Zugriffsfunktionen sind für alle Speichergrößen gleich, sodass problemlos auch andere Karten eingesetzt werden können. Der Speicher ist in Seiten bzw. Pages à 64 Bytes organisiert, im vorliegenden Falle in 256 Seiten. Die Adresse ist auf den Wert 0x50 festgelegt und nicht veränderbar. Da in der Regel nur ein Lesegerät am Bus angeschlossen ist sollte dies kein Problem darstellen.



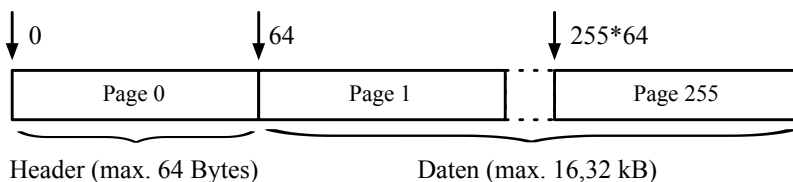
**Abb. 9.14.** Adressregister des AT24C128SC.

Nach der Übertragung einer 16-Bit-Adresse für die zugehörige Seite nach dem I<sup>2</sup>C-Adressbyte können für diese Seite bis zu 64 Bytes geschrieben werden. Dabei ist Vorsicht geboten: Werden mehr als 64 Byte geschrieben, so findet ein sog. *Roll-Over* innerhalb der Seite statt. Die zu Beginn gespeicherten Bytes werden dann erneut geschrieben bzw. überschrieben. Ein Lesen hingegen ist über den gesamten Bereich möglich. Die interne Adresse wird dabei nach jedem Zugriff auf ein einzelnes Daten-Byte (**Current Address Read**) oder eine Sequenz (**Sequential Read**) erhöht. Die Adresse kann zuvor mit einem Schreibvorgang ohne Datentransfer gesetzt werden.

Neben den genannten Zugriffsmethoden können Daten-Bytes auch zufällig ausgelesen werden. Dafür ist zunächst eine Dummy-Adresse zu senden, im Anschluss wird ein zufälliger Wert gelesen. Abbildung 9.15 zeigt die Speicheraufteilung der Karte und eine für die Klasse **IICChipcard** gewählte Repräsentationsform mit Header- und Datenteil.

### 9.7.2 Die Klasse **IICChipcard**

Folgende Überlegungen fließen in die Konzeption der Schnittstelle ein: Die Klasse **IICChipcard** soll I<sup>2</sup>C-Module vom Typ einer SmartCard repräsentieren und abstrakte Zugriffsfunktionen bereitstellen. Hier stellt sich zunächst die Frage, in welcher Form die Daten auf der Karte hinterlegt werden sollen. Je nach Art der zu speichernden Daten sind vielfältige Möglichkeiten denkbar. Sinnvoll ist es in jedem Fall, vor den eigentlichen Daten Informationen über die Karte selbst sowie über den enthaltenen Datentyp abzulegen, um die Chipkarten dieses Typs eindeutig identifizieren zu können. Hiermit kann im regulären Betrieb eine Überprüfung des Kartentyps erfolgen, um nicht fälschlicherweise die Chipkarte einer Bank oder einer Krankenkasse zu überschreiben. Diese Informationen werden in einem *Header* enthalten sein, welcher die erste Seite des Speichers belegt – 64 Byte sollten hierfür ausreichen (vgl. Abbildung 9.15).



**Abb. 9.15.** Speicherorganisation eines AT24C128SC-EEPROM-Chips: 16 384 Bytes Gesamtspeicher, organisiert in Seiten (*Pages*) zu je 64 Byte. In der Klasse **IICChipcard** wird die erste Seite für Header-Informationen verwendet, die restlichen Seiten stehen für Nutzdaten zur Verfügung.

Für den Header wurde in **Chipcard.h** die nachfolgende Struktur definiert, um Informationen über die Karte zu speichern. Falls benötigt, kann diese



zusätzlich um Informationen über den verwendeten Datentyp bzw. die Größe der Daten erweitert werden:

```
typedef struct {
    char type_id[10+1];    // ID for cards of this type
    char card_number[10+1]; // individual card number
    char descr[30+1];      // description
} ChipcardHeader_t;
```

Je nach Anwendungsfall sind die Kartendaten sehr unterschiedlich, sodass in dieser Klasse keine generische Struktur vorgeschlagen wird. Dies soll durch die Definition einer Struktur in der Anwendung selbst geschehen. Die Kartendaten werden direkt über die API bezogen bzw. an diese weitergeleitet und nicht in der Klasse zwischengespeichert. Als einzige Membervariable wird die Header-Information vorgehalten:

```
ChipcardHeader_t m_Header;    // header object
```

Die folgenden Methoden stehen dem Anwender als API zur Verfügung:

**IICChipCard(IICBus& bus, int addr, string name);**

Legt ein Objekt vom Typ **IICChipCard** an. Neben den Standardangaben für Bus-Objekt **bus**, Adresse **addr** und Bezeichnung **name** sind keine weiteren Informationen notwendig. Über **setIsRemovable()** wird angegeben, dass es sich bei diesem Modul um eine steckbare I<sup>2</sup>C-Komponente handelt, welche nicht immer am Bus vorhanden ist. Eine Fehlerkorrektur und eine automatische Abschaltung wird damit unterbunden.

**int readSCHeader();**

Liest den Header der Karte und liefert den Wert 0 zurück, falls die Typ-ID der Karte mit der Definition in **type\_id** übereinstimmt. Im Fehlerfall wird **-1** zurückgegeben.

**void printSCHeader();**

Liest den Header der Karte aus und gibt die Informationen auf der Standardausgabe aus.

**int formatSCHeader(const char\* card\_number, const char\* descr);**

Schreibt die Header-Informationen mit Kartenummer **card\_number** und Beschreibung **descr** auf eine neue Karte. Diese Funktion ist mit Vorsicht zu verwenden und sollte nur auf eindeutig bekannte Chipkarten angewandt werden. In der Applikation ist eine entsprechende Absicherung gegenüber vorschnellem Formatieren vorzusehen. Im Erfolgsfall wird 0 zurückgegeben, im Fehlerfall **-1**.

**int readSCData(char\* dest, unsigned long size);**

Liest **size** Daten-Bytes von der Karte und legt diese an der Stelle **dest** ab. Im Erfolgsfall wird die Anzahl gelesener Daten-Bytes zurückgegeben, sonst **-1**.

```
int writeSCData(char* source, unsigned long size);
```

Wie `readSCData()`, hier aber schreibend mit Bezug der Daten von `source`. Das Schreiben erfolgt seitenweise mit mehrmaliger Adressierung der Einzelseiten, da hierbei nicht wie beim Lesen ein *roll-over* zur nächsten Seite stattfindet. Im Erfolgsfall wird die Anzahl geschriebener Daten-Bytes zurückgegeben, sonst `-1`.

```
bool isAvailable();
```

Führt einen Dummy-Zugriffsversuch auf die Karte durch und überprüft so, ob diese eingesteckt ist. Diese Routine kann verwendet werden, falls keine Auswertung des entsprechenden Pins der Kontaktierungseinrichtung möglich ist.

Im Beispielverzeichnis `chipcard` veranschaulichen die zwei Beispiele `read_chipcard.cpp` und `write_chipcard.cpp` die Verwendung der Klasse in einer eigenen Anwendung. Die Formatierungsanweisung ist sicherheitshalber in `write_chipcard.cpp` zunächst deaktiviert und muss einkommentiert werden, um initial eine neue, leere Chipkarte zu beschreiben:

```
//#define WRITE_HEADER
```

Falls die Lese- und Schreibzugriffe mit verschiedenen Host-Rechner-Architekturen durchgeführt werden, so spielt auch hier – wie bei jeder Übertragung von Ganzzahlen als Binärdaten – die Byte-Reihenfolge eine wichtige Rolle. Abschnitt 13.2.5 im Kapitel zur Netzwerkkommunikation widmet sich dieser Problematik.

Eine Anmerkung: Für die Klasse `IICChipCard` bedeutet es keinen Unterschied, welcher Art die Daten auf der Karte sind, da nur reine Zeichenketten abgelegt werden. Zugriffs- und Manipulationsmethoden werden im Anwendungsprogramm implementiert. Der folgende Abschnitt zeigt, wie sensible Daten durch eine AES-Verschlüsselung abgesichert werden können.

### 9.7.3 AES-Verschlüsselung

Wenn auf einer Chipkarte sensible Daten gespeichert werden, so ist es sinnvoll, diese verschlüsselt abzulegen. Unter vielen möglichen Verschlüsselungsverfahren hat sich in den letzten Jahren der sog. *Advanced Encryption Standard (AES)* in vielen Anwendungen der Kommunikationstechnik und Datenhaltung durchgesetzt und wird unter anderem auch in WPA2<sup>13</sup>, SSH<sup>14</sup>, und RAR<sup>15</sup> verwendet.

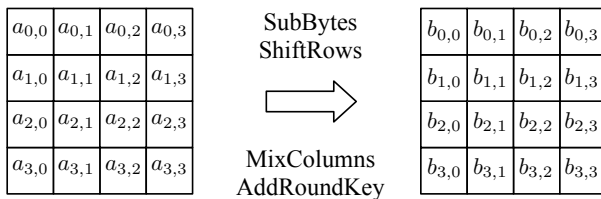
<sup>13</sup> *Wi-Fi Protected Access 2*, Implementierung eines Sicherheitsstandards für Funknetzwerke.

<sup>14</sup> *Secure Shell*, verschlüsseltes Netzwerkprotokoll.

<sup>15</sup> *Roshal Archive*, Datenkompressionsalgorithmus.

Der AES-Algorithmus soll lediglich in den Grundzügen vorgestellt werden, um die Implementierung der Klasse AES im nächsten Abschnitt zu erklären. Für weiterführende Informationen sei auf die offizielle AES-Spezifikation verwiesen [NIST 08]. AES ging als Gewinner eines vom *National Institute of Standards and Technology (NIST)* ausgeschrieben, offenen Wettbewerbs hervor und löste den bis dahin verwendeten *Data Encryption Standard (DES)* als neuen Standard ab. AES gilt als sehr sicher und ist in den USA auch für die Verschlüsselung von Dokumenten mit hoher Geheimhaltungsstufe zugelassen.

AES ist ein iterativer Blockchiffrier-Algorithmus, bei dem Datenblöcke von 128 bit mit einem Schlüssel der Länge 128, 192 oder 256 bit verschlüsselt werden. Er basiert auf dem *Rijndael-Algorithmus* und stellt durch die Datenblocklänge eine spezialisierte Form dar. In der Originalversion ist diese mit 128, 160, 192, 224 oder 256 bit variabel gehalten. Ein 128-Bit-Datenblock lässt sich als Matrix mit  $4 \times 4$  Bytes darstellen. Auf jeden Datenblock werden nacheinander bestimmte Transformationen angewandt (vgl. Abbildung 9.16). Diese Transformationen hängen dabei nicht vom vollständigen Originalschlüssel ab, sondern von Teilen davon.



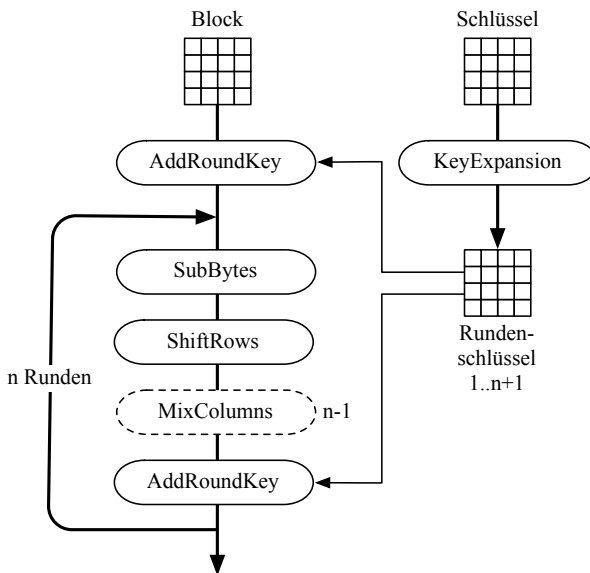
**Abb. 9.16.** Vier mögliche AES-Transformationsschritte für einen  $4 \times 4$ -Datenblock.

Die AES-Verschlüsselung läuft in mehreren Runden ab, wobei die Anzahl  $n$  von der gewählten Verschlüsselungstiefe abhängt (vgl. Abbildung 9.17). Mit einem Schlüssel der Größe 128 werden 10 Verschlüsselungsrunden durchgeführt, bei einer Länge von 192 oder 256 bit sind dies 12 bzw. 14 Runden. In einem initialen Schritt werden aus dem Schlüssel  $(n + 1)$  sog. Rundenschlüssel extrahiert (*KeyExpansion*). Der Datenblock wird dann zunächst mit dem Rundenschlüssel verknüpft (*AddRoundKey*), bevor mit dem Schleifendurchlauf begonnen wird. Während jeder Runde werden nacheinander die folgenden Operationen auf dem zu verschlüsselnden Datenblock angewandt:

1. *SubBytes*: In jeder Runde wird zunächst für alle Felder im Block ein Äquivalent in der S-Box gesucht. Die Daten werden damit monoalphabetisch verschlüsselt.
2. *ShiftRow*: Bei dieser Transformation werden die Zeilen der  $4 \times 4$ -Matrix um eine bestimmte Anzahl von Spalten nach links verschoben. Überlaufende

Zellen werden erneut rechts beginnend fortgesetzt. Bei einer Blocklänge von 128 bit beträgt die Anzahl der Verschiebungen 0, 1, 2 und 3.

3. *MixColumn*: Nach *ShiftRow* werden nun die Spalten vermischt. Jede Zelle einer Spalte wird dazu mit einer Konstanten multipliziert. Anschließend werden die Ergebnisse bitweise XOR-verknüpft. Durch diese Transformation tritt jedes Byte einer Spalte in Wechselwirkung mit jedem anderen Byte der Spalte. Der mathematische Zusammenhang ist in [NIST 08] ausführlich erklärt.
4. *AddRoundKey*: In der Initialisierung und am Ende jeder Verschlüsselungsrunde wird *AddRoundKey* ausgeführt. Hierbei wird der Block mit dem aktuellen Rundenschlüssel bitweise XOR-verknüpft. Dies ist die einzige Funktion in AES, in die der Benutzerschlüssel eingeht.



**Abb. 9.17.** AES-Verschlüsselungsprozess.

In der letzten Verschlüsselungsrunde wird auf die Transformation *MixColumns* verzichtet. Für die Durchführung einer Verschlüsselung wird in umgekehrter Reihenfolge vorgegangen. Die AES-Implementierung ist im Vergleich zu anderen Algorithmen sehr performant und lässt sich auch auf kleineren Prozessoren bis hin zu 8-Bit-Mikrocontrollern schnell durchführen. Der RAM-Speicherbedarf ist durch die blockweise Behandlung relativ gering. Bei stark beschränkten Ressourcen können die einzelnen Rundenschlüssel auch nacheinander berechnet werden.

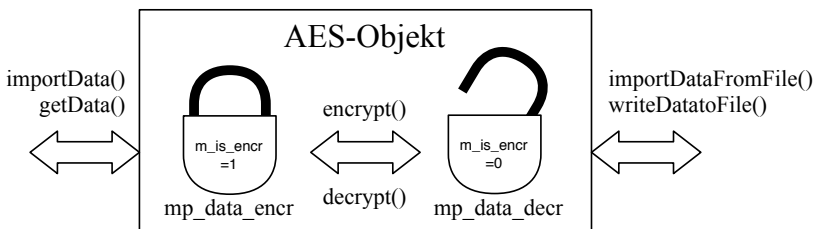
Im folgenden Abschnitt wird der AES-Algorithmus in einer Klasse AES implementiert, um über eine allgemein gehaltene Schnittstelle aus Dateien oder Speicherbereichen verschlüsselte Datenobjekte zu erzeugen.

#### 9.7.4 Die Klasse AES

Die Klasse AES repräsentiert verschlüsselte oder unverschlüsselte Datenobjekte und stellt Methoden zur Manipulation bereit. Als Datenquellen kommen je nach Anwendung Dateien oder Speicherbereiche in Frage. Die bereitgestellten Daten können bereits verschlüsselt sein oder auch im Klartext vorliegen.

Für eine möglichst übersichtliche Gestaltung der API wird keine spezielle Ein- und Ausgabemethode für ver- oder entschlüsselte Daten implementiert, sondern ein Großteil der Logik in die Klasse integriert. Die AES-Quelltext-Dateien sind unter `<embedded-linux-dir>/src/tools/` abgelegt.

Die Betrachtungsweise ist wie folgt: Ein Objekt vom Typ AES wird als abstrakter Datenspeicher angesehen, in welchen ver- oder entschlüsselte Daten geladen werden können (vgl. Abbildung 9.18). Eine Platzierung der Daten in den Membervariablen `mp_data_encr` oder `mp_data_decr` geschieht automatisch. Der Anwender muss lediglich die Datenquelle in Form einer Speicherstelle oder einer Datei angeben, nicht aber den Verschlüsselungszustand.



**Abb. 9.18.** Repräsentation von verschlüsselten und unverschlüsselten Daten in Objekten der Klasse AES.

Durch das Ver- oder Entschlüsseln eines AES-Objektes werden die internen Datenstrukturen abhängig vom aktuellen Zustand `m_is_encr` ineinander überführt. Das Objekt ist somit zu jedem Zeitpunkt entweder *geschlossen* oder *geöffnet*. Erfolgt ein Zugriff über `getData()` oder `writeDataToFile()`, so werden Daten wiederum abhängig vom aktuellen Zustand geliefert. Für eine Unterscheidung des Zustandes wird den verschlüsselten Daten die Struktur `AESHeader` vorangestellt. Dies ist insbesondere wichtig, um die korrekte Größe der Originaldaten mitzuführen, da die Größe der verschlüsselten Daten immer ein Vielfaches von 16 Bytes beträgt.

```
typedef struct {
    char uin[UIN_SIZE+1];          // id number to identify encr. data
    unsigned int decr_size;         // size of decr. data
    unsigned int encr_size;         // size of encr. data
} AESHeader;
```

Die Zeichenkette `encryption_uin` zu Beginn dient der Identifikation eines verschlüsselten Datensatzes. Auf die für die unterschiedlichen Transformation notwendigen Membervariablen soll an diese Stelle nicht weiter eingegangen werden. Als API stehen dem Anwender folgende Funktionen zur Verfügung:

`AES();`

Erzeugt Objekte vom Typ AES und benötigt keine weiteren Angaben.

`int encrypt();`

Verschlüsselt die an der Stelle `mp_data_decr` abgelegten Daten, falls unverschlüsselte Daten vorhanden sind und zuvor noch keine Verschlüsselung stattgefunden hat. Die Daten werden in `mp_data_encr` abgelegt. Im Erfolgsfall wird 0 zurückgegeben, sonst `-1`.

`int decrypt();`

Entschlüsselt die an der Stelle `mp_data_encr` abgelegten Daten, falls verschlüsselte Daten vorhanden sind und zuvor noch nicht entschlüsselt wurde. Die Daten werden in `mp_data_decr` abgelegt. Im Erfolgsfall wird 0 zurückgegeben, sonst `-1`.

`void importData(char* p_data, unsigned int size);`

Importiert `size` Bytes (un-)verschlüsselter Daten, die an der Speicherstelle `p_data` abgelegt sind. Der Verschlüsselungszustand wird automatisch bestimmt.

`char* getData();`

Liefert einen Zeiger vom Typ `char` auf `mp_data_encr` oder `mp_data_decr`. Welche Daten ausgegeben werden, hängt vom aktuellen Zustand ab: Ist dieser *geschlossen*, so werden verschlüsselte Daten inklusive Header geliefert, sonst unverschlüsselte Daten.

`int getEncryptedSize();`

Gibt die Größe des verschlüsselten Dateninhaltes zurück. Liegen keine Daten vor, so wird 0 zurückgeliefert.

`int getDecryptedSize();`

Gibt die Größe des unverschlüsselten Dateninhaltes zurück. Liegen keine Daten vor, so wird 0 zurückgeliefert.

`int importDataFromFile(string fname);`

Liest (un-)verschlüsselte Daten aus der Datei `fname`. Im Erfolgsfall wird 0 zurückgegeben, sonst `-1`.

```
int writeDataToFile(string fname);
```

Schreibt abhängig vom Zustand des Objekts (un-)verschlüsselte Daten in eine Datei `fname`. Im Erfolgsfall wird 0 zurückgegeben, sonst `-1`.

```
int isEncrypted();
```

Liefert den aktuellen Zustand als verschlüsselt (1) oder unverschlüsselt (0) zurück.

```
unsigned long decrToEncrSize(unsigned long size);
```

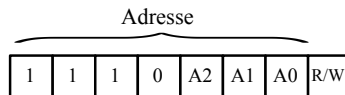
Rechnet die Länge von `size` Bytes unverschlüsselter Daten in die zugehörige Größe für die verschlüsselte Form um.

## 9.8 I<sup>2</sup>C-Bus-Erweiterung über Multiplexer

Der relativ kleine Adressbereich eines I<sup>2</sup>C-Busses kann mit sog. Multiplexer-Bausteinen erweitert werden. Der Hauptbus wird dazu in mehrere umschaltbare Unterbusse aufgesplittet. In der Regel werden diese Multiplexer dann benötigt, wenn mehr als acht I<sup>2</sup>C-Bausteine gleichen Typs an einem Bus betrieben werden sollen. Eine sinnvolle Gruppierung der Bausteine hinsichtlich ähnlicher Abtastzeiten kann helfen, die Anzahl der Umschaltvorgänge und damit den Overhead gering zu halten.

### 9.8.1 Philips PCA9548 I<sup>2</sup>C-Multiplexer

Der PCA9548-Baustein von Philips ist ein I<sup>2</sup>C-Multiplexer mit acht umschaltbaren Kanälen. Über ein Kontrollregister kann der Schaltzustand festgelegt und ein *Downstream*-Kanal an den SCL/SDA-*Upstream* angeschlossen werden. Die Adresse des PCA9548 wird über drei Bits eingestellt (vgl. die Basisadresse in Abbildung 9.19). Nach einem Neustart ist zunächst kein Kanal ausgewählt. Die Auswahl erfolgt nach der Übertragung des Kontrollregister-Bytes. Neben dem Lesen des Kontrollregisters ist dies die einzige Zugriffsfunktion auf einen PCA954x.



**Abb. 9.19.** Adressregister des PCA9548.

Durch Zuschalten mehrerer I<sup>2</sup>C-Kanäle kann sich die Kapazität der I<sup>2</sup>C-Busleitung erheblich erhöhen. Dieser Umstand ist bei einer Auslegung der Gesamtlänge zu berücksichtigen. Die Bausteine könnten zwar grundsätzlich bis

zum Erreichen der Gesamtkapazität kaskadiert werden, zu bedenken ist aber auch die Organisation. Für die Adressvergabe und die Identifikation erlaubter Multiplexer-Schaltzustände ist eine genaue Kenntnis der Bustopologie notwendig. Aus Gründen der Übersichtlichkeit liegt es nahe, die Verwendung der Multiplexer auf den Hauptbus zu beschränken. Von CC-Tools kann ein fertiges Modul bezogen werden, einzelne I<sup>2</sup>C-Bausteine sind bei Reichelt erhältlich (vgl. [CC-Tools 07], [Reichelt 08] und Anhang E).

### 9.8.2 Die Klasse IICMultiplexer

Die Klasse `IICMultiplexer` soll Bausteine der Typen PCA954x repräsentieren, wobei das „x“ für die Anzahl der Kanäle steht. Zum Vorhalten dieses Wertes und des aktuell ausgewählten Kanals dienen zwei Membervariablen:

```
int m_max_channel;
int m_channel;
```

In der sehr überschaubaren Klasse stehen lediglich der Konstruktor mit drei Memberfunktionen als API zur Verfügung. Eine Initialisierung ist nicht notwendig:

```
IICMultiplexer(IICBus& bus, int addr, string name, int
max_channel);
```

Erstellt ein Objekt vom Typ `IICMultiplexer` unter Angabe der Standardparameter für das Bus-Objekt `bus`, der Adresse `addr` und einer Bezeichnung `name`. Zusätzlich wird die maximale Anzahl an Multiplexerkanälen `max_channel` übergeben.

```
int getMaxChannel();
```

Gibt die maximale Anzahl an Kanälen zurück.

```
int getCurrChannel();
```

Liefert im Erfolgsfall den momentan eingestellten Kanal, sonst `-1`.

```
int setChannel(int ch);
```

Setzt den aktiven Kanal auf `ch`, falls dieser nicht bereits ausgewählt ist.

Das Beispiel `temp_sensor_multi` veranschaulicht die Verwendung. Die Multiplexer-Objekte werden dem Bus-Objekt vom Typ `IICBus` hinzugefügt wie andere Module auch. Über einen Aufruf von `makeSubModule()` wird dem Objekt `bus` zusätzlich signalisiert, dass das I<sup>2</sup>C-Modul `modul` nicht am Hauptbus, sondern an einem Unterbus angeschlossen ist. Als Parameter wird der Kanal `ch` des Multiplexer-Bausteins `mpx` übergeben:

```
bus.makeSubModule(<modul>, <mpx>, <ch>);
```



Alle Aufrufe der `IICMultiplexer`-Methoden werden innerhalb der Busklasse durchgeführt und vor dem Anwender verborgen. Das heißt konkret, dass vor jedem Schreib- bzw. Lesevorgang überprüft wird, ob der Kanal für den aktuell anzusprechenden Baustein ausgewählt ist.

In der aktuellen Implementierung ist nur der direkte Anschluss der Multiplexer-Bausteine am Hauptbus erlaubt. Weitere Verzweigungen von Unterbussen werden (noch) nicht unterstützt. Die Beschränkung auf nur einen aktiven Zweig<sup>16</sup> zu jedem Zeitpunkt erfordert ein häufiges Umschalten. Der Vorteil ist jedoch, dass hiermit auch eine einfache Überprüfung der zulässigen Baustein-Kombinationen möglich ist. Die Beispiel-Implementierung zeigt die Ansteuerung von fünf Temperatursensoren, wobei ein Sensor an den Hauptbus angeschlossen ist und jeweils zwei über verschiedene Multiplexerkanäle Anschluss finden.

---

<sup>16</sup> Nur ein Kanal aller vorhandenen Multiplexer kann geschaltet sein.

## USB-Komponenten

### 10.1 Einführung

Die Theorie zum *Universal Serial Bus (USB)* wurde im Rahmen der seriellen Bussysteme bereits in Kapitel 8 behandelt, sodass in diesem Kapitel lediglich praktische Beispiele für Schnittstellenerweiterungen über USB vorgestellt werden. USB hat in den letzten Jahren den PCMCIA-Bus als Erweiterungsschnittstelle zunehmend ersetzt. USB-Adaptermodule für WLAN, GPS, Audio oder Bluetooth sind bereits zwischen zehn und zwanzig Euro erhältlich und für Embedded-Systeme ohne PCI- oder PCI-Express-Schnittstelle oftmals die einzige Möglichkeit der Erweiterung.

Bevor in den folgenden Abschnitten die Inbetriebnahme verschiedener USB-Adapter besprochen wird, sollten zunächst die Voraussetzungen für einen Betrieb von USB-Geräten geschaffen werden. Mit dem Programm `lsusb` (enthalten im Paket `usbutils`) können angesteckte USB-Geräte aufgelistet werden. Die Sichtbarkeit in dieser Liste ist die Grundvoraussetzung, um ein USB-Gerät nutzen zu können. Während bei Debian die notwendigen Treiber und auch das Paket `usbutils` üblicherweise installiert sind, so ist dies bei OpenWrt nicht der Fall. Um sicher zu gehen, dass die volle USB-Unterstützung vorhanden ist, sollten bei OpenWrt die folgenden Pakete installiert sein:

```
$ ipkg list_installed | grep usb
kmod-usb-audio - 2.6.22-brcm47xx-1 -
kmod-usb-core - 2.6.22-brcm47xx-1 -
kmod-usb-ohci - 2.6.22-brcm47xx-1 -
kmod-usb-uhci - 2.6.22-brcm47xx-1 -
kmod-usb2 - 2.6.22-brcm47xx-1 -
libusb
usbutils
```

Wenn unter OpenWrt ein Aufruf von `lsusb -t` eine Fehlermeldung erzeugt, so kann der fehlende Eintrag im `/proc`-Verzeichnis durch folgenden Aufruf ergänzt werden:

```
$ mount -t usbfs none /proc/bus/usb
```

Ein eingesteckter Adapter sollte nun bei einem `lsusb`-Aufruf erscheinen:

```
$ lsusb
Bus 003 Device 001: ID 0000:0000
Bus 002 Device 001: ID 0000:0000
Bus 001 Device 005: ID 0d8c:000c C-Media Electronics, Inc. Audio Adapter
Bus 001 Device 001: ID 0000:0000
```

Für Puppy existiert leider kein Befehl `lsusb`. Hier muss der Anwender das Tool entsprechend nachinstallieren (vgl. Abschnitt 14.2) oder sich über die Systemeinträge im `/proc`-Verzeichnis behelfen:

```
$ cat /proc/bus/usb/devices
```

Im folgenden Kapitel werden zunächst verschiedene USB-Adapter für Audio, WLAN und Bluetooth vorgestellt und deren Einbindung in das System erläutert. In der zweiten Hälfte des Kapitels wird ein GPS-Modul verwendet, um die Position mobiler Geräte zu bestimmen. Danach wird die Erweiterung des bei Embedded-Systemen oft begrenzten Festspeicherplatzes durch USB-Speichersticks diskutiert. Herkömmliche RS-232-USB-Adapter verwenden oft einen PL2303-Chip von Prolific. Für die Einbindung eines solchen Adapters sei auf Abschnitt 10.5 verwiesen, da im GPS-Modul der gleiche Chipsatz verwendet wird.

Die Quelltext-Beispiele zu dem vorliegenden Kapitel sind im Unterverzeichnis `<embedded-linux-dir>/examples/usb/` zu finden.

## 10.2 USB-Audioanbindung: MP3-Player und Sprachausgabe

Die sog. *Advanced Linux Sound Architecture* [ALSA 08] stellt eine Sammlung von Treibern, Bibliotheken und Werkzeugen zur Audiounterstützung unter Linux dar. Viele Programme wie *Madplay* oder *Flite* greifen auf die ALSA-Schnittstelle zu. In den ALSA-Treibern wird auch die Schnittstelle zum etwas in die Jahre gekommenen *Open Sound System (OSS)* unterstützt, welches inzwischen fast komplett verdrängt wurde, aber von manchen Programmen noch immer benötigt wird. Im vorliegenden Abschnitt wird ein herkömmlicher USB-Audio-Adapter verwendet, der zum Preis von ca. 10–15 EUR erhältlich ist (vgl. Tabelle E.1).

Die ALSA-Unterstützung ist bei Debian-Distributionen oft bereits in den Kernel integriert. Eine Ausnahme bildet die von VisionSystems für den Alekto gepatchte Distribution. Hier müssen in der Kernelkonfiguration die Module

*USB Audio/Midi Driver*<sup>1</sup> und *OSS PCM (digital audio) API*<sup>2</sup> eingebunden werden (vgl. auch Abschnitt 5.7). Unter Puppy ist die Audio-Unterstützung von Haus aus vorhanden.

Für OpenWrt wird ein rudimentärer Linux-Kernel verwendet, der nur mit den notwendigsten Treibern ausgestattet ist. Entsprechend ist hier die ALSA-Unterstützung in Form weiterer Kernelmodule nachzuinstallieren. Für OpenWrt werden dazu die Pakete `kmod-usb-audio` und `kmod-sound-core` benötigt.

Nachdem die ALSA-Unterstützung gesichert ist, können nun auch die Konfigurationsdateien des ALSA-Treibers und die ALSA-Werkzeuge installiert werden. Für Debian sind diese in den folgenden Paketen enthalten:

```
$ apt-get install alsa-base alsa-utils
```

Davon abhängende Module wie bspw. die ALSA-Bibliothek werden von *Aptitude* aufgelöst und installiert. Unter OpenWrt erfolgt die Installation durch den Aufruf:

```
$ ipkg install alsa-utils
$ ipkg install alsa-lib
```

Bestandteil der Werkzeuge ist auch das Programm *Alsamixer*. Mit diesem lassen sich z. B. Ein- und Ausgangspegel einer Soundkarte über die Kommandozeile einstellen. Mit der ALSA-Bibliothek ist die Grundlage vorhanden, weitere Software zu verwenden: Das Programm *Madplay* ist ein schlankes Tool, um Audiodateien vom Typ WAV oder MP3 abzuspielen. Im Gegensatz zu anderen Playern werden in Madplay alle Berechnungen in Festkommaarithmetik durchgeführt, was real einer Verwendung von 32-Bit-Ganzzahlberechnungen entspricht. Relevant ist dies, um auf Rechnern ohne Gleitkommaeinheit<sup>3</sup> eine Dekodierung effizient umsetzen zu können. Madplay ist auf Puppy bereits installiert, für Debian und OpenWrt stehen gleichnamige Pakete zur Verfügung.

Beim Start von `madplay` zeigt sich, dass auf einer NSLU2 mit 266 MHz und ohne FPU lediglich 8–10 % der Rechnerauslastung auf den Player entfallen:

```
$ madplay test.mp3 &
$ bg
$ top
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2448	joa	15	0	2504	872	736	S	8.8	2.9	0:02.42	madplay
2443	joa	15	0	2300	1072	848	R	1.3	3.6	0:00.74	top
1	root	15	0	1896	620	540	S	0.0	2.1	0:04.99	init
2	root	34	19	0	0	0	S	0.0	0.0	0:03.17	ksoftirqd/0

Eine synthetische Sprachausgabe auf dem Embedded-PC kann mit dem Programm *Flite*<sup>4</sup> [Flite 08] erzeugt werden. Tatsächlich sind sowohl Sprachqualität

<sup>1</sup> Zu finden unter *Device Drivers/Sound/ALSA/USB Devices*.

<sup>2</sup> Zu finden unter *Device Drivers/Sound/ALSA/*.

<sup>3</sup> Engl.: *Floating Point Unit (FPU)*.

<sup>4</sup> Zusammengesetzt aus der Bezeichnung *Festival-Lite*.

als auch Leistungsumfang eher bescheiden, und so wird bspw. nur eine Stimme in englischer Sprache unterstützt. Da umfangreiche Spracherzeugungssysteme wie die Kompletversion *Festival* sehr viel mehr Rechenleistung benötigen, liegt der Vorteil von **flite** in der effizienten Nutzung der Ressourcen. **flite** läuft auf leistungsschwachen Rechnern und ermöglicht auch auf diesen die parallele Bearbeitung weiterer Aufgaben.

Nach der Installation des Paketes **flite**<sup>5</sup> lässt sich über die Kommandozeile ein Text vorgeben, der von Flite synthetisiert wird:

```
$ flite -t "This is a test"
```

Auch wenn lediglich eine männliche Computerstimme mit etwas monotonem Klang ertönt, so lassen sich damit doch zumindest Ausgaben wie z. B. Warnmeldungen erzeugen, die je nach Einsatzgebiet praktischer sind als visuelle Hinweise. Eine Integration in eigene Anwendungen wird durch die Verwendung der Flite-Bibliothek möglich – hierzu muss das Paket **flite1-dev** installiert werden (Debian). Das Beispiel **audio** zeigt die Verwendung der Bibliothek und lässt sich nach der Übersetzung mit Angabe einer Textdatei starten:

```
$ ./read_text test.txt
```

## 10.3 USB-WLAN-Adapter

### 10.3.1 Grundlagen

Der erste und wichtigste Schritt bei der Installation eines USB-WLAN-Adapters ist die Suche nach einem passenden Treiber. Die Linux-Unterstützung für WLAN-Geräte ist zwar in den letzten Jahren immer besser geworden, die Verfügbarkeit eines quelloffenen Treibers ist aber noch immer keine Selbstverständlichkeit.

Zuerst sollte man probierhalber den WLAN-Adapter an einem Desktop-PC unter der aktuellen Ubuntu-Version einstecken. Mit etwas Glück wird er sofort erkannt, und die unterstützende (Debian)-Treiberversion kann nun auch für das Embedded Device gesucht werden. Wenn der Stick nicht erkannt wird, so können nun zumindest mittels **lsusb** Details zum Chipsatz in Erfahrung gebracht werden. Mit der ID und dem Namen des Chipsatzes wird man dann u. U. in einer der folgenden Datenbanken fündig:

```
http://www.linux-usb.org
http://www.linux-wlan.org/docs/wlan_adapters.html.gz
http://www.linux-wlan.com/download.shtml
```

<sup>5</sup> Dieses Paket kann bei vielen OpenWrt-Geräten nur bei Erweiterung durch zusätzlichen externen Speicher installiert werden (vgl. Abschnitt 10.6.2 am Ende).

<http://prism54.org>  
<http://madwifi.org>  
[http://www.hpl.hp.com/personal/Jean\\_Tourrilhes/Linux/Wireless.html](http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Wireless.html)

Besonders wertvoll ist hierbei die letzte Quelle bzw. die dortigen PDF-Dokumente (das „Linux Wireless LAN Howto“). Weiterhin hilfreich ist ein Linux-Standardwerk wie z. B. [Ayaz 06].

Noch immer gibt es keine zentrale Datenbank zur Klärung der Treiberunterstützung für solche Geräte und auch die genannten Quellen sind per se immer ein wenig veraltet. Mittlerweile bieten aber auch viele Hardware-Hersteller wie Netgear u. a. auf der Produkt-Website Linux-Treiber an.

Falls zu dem Gerät kein Linux-Treiber erhältlich ist, so ist oft dennoch zumindest unter der x86-Architektur eine Anbindung möglich. Unter der Voraussetzung, dass eine Windows-Treiberversion als inf- und sys-Datei verfügbar ist, kann der sog. NDISWrapper<sup>6</sup> zum Einsatz kommen. Der Wrapper implementiert die NDIS API und die notwendigen Windows-Kernel-Funktionen und linkt den Windows-Treiber dynamisch zu dieser Implementierung. Vgl. auch:

<http://wiki.ubuntuusers.de/WLAN/ndiswrapper>  
<http://ndiswrapper.sourceforge.net>

Zu beachten ist, dass der NDISWrapper nur auf x86- oder x86\_64-Systemen lauffähig ist. Auf Hardware-Plattformen wie dem MicroClient Sr. können hiermit entsprechend auch exotischere WLAN-Geräte angebunden werden.<sup>7</sup> Für ARM-basierte Plattformen wie die NSLU2 scheidet diese Vorgehensweise aber leider aus.

Zu den unter Linux verfügbaren WLAN-Tools wie iwconfig, iwpriv, iwlist und wlanctl-ng (letzteres speziell für Prism-Chipsätze) vgl. die Manpages.

### 10.3.2 Netgear MA111 unter Puppy

Im folgenden Text wird die Inbetriebnahme eines USB-WLAN-Adapters unter Puppy Linux erklärt. Folgende Hardware wird verwendet: NorhTec MicroClient Sr. + NetGear Wireless USB Adapter MA111. Die Betriebssystemversion lautet Puppy Linux 3.01.

<sup>6</sup> NDIS steht für Network Driver Interface Specification. Es handelt sich um eine API für Netzwerk-Geräte, die gemeinsam von Microsoft und 3Com entwickelt wurde.

<sup>7</sup> Eine Anmerkung: Der NDISWrapper unterstützt auch PCI-, PCMCIA- und Mini-PCI-Geräte.

Zuerst wird der Stick bei gebootetem Rechner eingesteckt. Das Programm `lsusb` liefert folgende Ausgabe (zu einer u. U. erforderlichen Nachinstallation dieses Tools vgl. Abschnitt 14.2):

```
$ ./downloads/root/my-applications/bin/lsusb
Bus 003 Device 001: ID 0000:0000
Bus 002 Device 003: ID 15ca:00c3 Textech International Ltd. Mini Optical
Mouse
Bus 002 Device 001: ID 0000:0000
Bus 001 Device 004: ID 0846:4110 NetGear, Inc. MA111(v1) 802.11b Wireless [
Intersil Prism 3.0]
Bus 001 Device 003: ID 046a:0023 Cherry GmbH Cymotion Master Linux Keyboard
Bus 001 Device 001: ID 0000:0000
```

Die Ausgabe Intersil Prism 3.0 zeigt bereits den Chipsatz an, womit nun auch der erforderliche Treiber bekannt ist. Die im vorangegangenen Text aufgeführten Datenbanken liefern weitere Informationen zur Unterstützung. Bei Sichtung der Quellen zeigt sich, dass der Chipsatz grundsätzlich unterstützt wird. Folgender Befehl kontrolliert die Einbindung:

```
$ dmesg | grep -i usb
```

Eine Durchsicht der Ausgabe zeigt, dass das Modul `prism2_usb` tatsächlich geladen wurde. Der Adapter sollte also grundsätzlich funktionieren. Zuvor muss allerdings für die Verwendung von DHCP die DHCP-Tabelle im Netzwerk-Router entsprechend um einen Eintrag ergänzt werden (die MAC-Adresse steht meist auf dem Stick). Noch einfacher geht es, wenn das Gerät bereits am Router angemeldet ist. Ist im laufenden Betrieb später nochmals die MAC-Adresse erforderlich, so ist sie wie auch die IP durch folgende Eingabe zu ermitteln:

```
$ ifconfig -a
```

Nach der Konfiguration des Routers kann die Netzwerkeinrichtung auf dem MicroClient erfolgen: Per Connect-Symbol auf dem Desktop gelangt man in den Einricht-Dialog. Hier ist auszuwählen: Connect to Network by Network Interface. Autoload USB, dann: wlan0, Wireless. Im erscheinenden Dialog sind die notwendigen Einstellungen hinsichtlich SSID und Schlüssel einzutragen. Jetzt klicken: Save, dann: Use this profile. Der Scan-Button zeigt nun bereits verfügbare Netze an. Nun einen Dialog zurück, dort ist auszuwählen: Auto-DHCP. Die notwendigen Einstellungen schließen mit Save und Done. Wenn der USB-WLAN-Adapter beim Booten bereits eingesteckt ist, so wird er von nun an vom System automatisch gefunden und initialisiert. Wenn er aber im laufenden Betrieb eingesteckt wird, so ist nochmals per Connect-Dialog der Treiber zu laden (Autoload USB).

Die beschriebene Vorgehensweise funktioniert grundsätzlich auch für modernere Sticks wie z. B. den Netgear WG111v2. Für diesen wird man über die genannten Links und Datenbanken bei <http://sourceforge.net> hinsichtlich einer Open-Source-Implementierung fündig.

Das Problem der ARM-Plattformen bleibt aber bestehen, da die Implementierungen u. U. auf x86-spezifische Funktionen setzen bzw. eine Floating Point

Unit (FPU) voraussetzen. Unsere Versuche, den Stick auf der NSLU2 zum Laufen zu bringen, sind jedenfalls bisher gescheitert.

### 10.3.3 Alternative: WLAN-Anbindung über Access Point

Im vorigen Abschnitt wurde die WLAN-Anbindung von Embedded-PCs über USB-WLAN-Adapter vorgestellt. Bei dieser Art der Anbindung treten zumindest auf Nicht-x86-Geräten häufiger Probleme auf. Es ist aber eine elegante Lösung möglich: Die Anbindung des Rechners über einen Access Point. Diese Geräte sind mittlerweile auch in kleiner Bauform verfügbar und liegen preislich in einer ähnlichen Klasse wie hochwertige USB-Adapter.<sup>8</sup>

Der WLAN-Zugang wird bei dieser Lösung über ein separates Gerät realisiert, wobei der Embedded-PC über eine herkömmliche Ethernet-Verbindung kommuniziert.

Der große Vorteil dieser universellen Lösung ist, dass sie mit jedem Embedded-PC, der eine Ethernetschnittstelle besitzt, funktionieren sollte. Die Nachteile sollen aber nicht verschwiegen werden: Da ein weiteres Gerät notwendig wird, ist diese Art der Anbindung nicht so energieeffizient. Weiterhin beansprucht diese Lösung mehr Bauraum als eine Anbindung über USB-WLAN-Adapter.

Wichtig ist, dass der Access Point im sog. Client-Mode (auch als Station-Mode bezeichnet) betrieben werden kann. Er hängt sich in diesem Modus an ein bestehendes WLAN-Netz an und stellt kein eigenes WLAN-Netz bereit. Oftmals lässt sich dieser Modus über einen externen Schalter festlegen. Über die voreingestellte LAN-IP kann die Konfiguration des APs über einen direkt angeschlossenen PC erfolgen (dieser muss sich im gleichen Subnetz befinden). Für den AP ist eine freie IP aus dem Hausnetz einzutragen, sodass dieser nach dem Aufbau einer WLAN-Verbindung ins Hausnetz nicht mit anderen Komponenten kollidiert. In der Regel kann über die Web-Oberfläche nach WLAN-Netzen gesucht werden, mit denen der AP verbunden werden soll.

Nach der Konfiguration und dem Test der WLAN-Verbindung kann der AP an den Embedded-PC angeschlossen werden. Für unsere Versuche haben wir hierzu die NSLU2 verwendet.

Die NSLU2 wird hinsichtlich der Netzwerkeinstellungen konfiguriert (statisch oder DHCP), als ob sie direkt am Hausnetz hängen würde. Der AP dient damit als Brücke und übernimmt weder Router- noch DHCP-Server-Funktionalität. Ein evtl. laufender DHCP-Server auf dem AP ist abzuschalten, da die NSLU2-IP sonst für die anderen Rechner im Hausnetz unerreichbar bleibt.

---

<sup>8</sup> Beispiel: Longshine LCS-WA5-45, ca. 35 EUR (vgl. Tabelle E.1).



## 10.4 USB-Bluetooth-Erweiterung

Die Funkanbindung via Bluetooth wird in vielen mobilen Geräten zur drahtlosen Kommunikation eingesetzt. Bekannte Beispiele sind die Anbindung von Headsets oder die Datensynchronisation. Aktuelle Treiber lassen eine einfache Verwendung von Bluetooth-USB-Sticks unter Linux zu, sodass sich für eine Integration in eigene Anwendungen vielfältige Möglichkeiten ergeben.

Der vorliegende Abschnitt beschäftigt sich zunächst mit den Grundlagen und verfügbaren Werkzeugen. Im Anschluss werden mehrere praktische Beispiele vorgestellt: Die Verwendung des Handys als Fernsteuerung und die Realisierung einer seriellen Funkstrecke.

### 10.4.1 Grundlagen

Bluetooth wurde für die drahtlose Kommunikation von Geräten über kurze Distanzen entwickelt und ist durch den Standard IEEE 802.15.1 spezifiziert. Üblicherweise nutzen Geräte wie Handys oder Handhelds, Computer oder auch Spielkonsolen diese Art der Kommunikation. Mit neueren Übertragungsstandards und höheren Datenraten soll es zukünftig auch möglich werden, Drucker oder Bildschirme via Bluetooth anzusteuern. In der aktuellen Bluetooth-Version 2.1 ist eine Datenrate bis 2,1 Mbit/s möglich, die auf maximal sieben aktive Verbindungen verteilt werden kann.

Für die Übertragung wird das 2,4-GHz-Band (2 400–2 483,5 MHz) verwendet, wobei der Frequenzbereich in insgesamt 79 Kanäle unterteilt ist. Die Sendestärke liegt je nach Klasse zwischen 1 mW und 100 mW und ermöglicht eine Reichweite zwischen 1 und 100 m (vgl. Tabelle 10.1.). Diese Angaben sind als maximal mögliche Werte unter idealen Bedingungen zu verstehen. In Gebäuden, bei Hindernissen in der Funkstrecke oder bei zu kleinen Antennen verkürzt sich die Distanz erheblich.

Klasse	Max. Leistung	Reichweite (im Freien)
Klasse 1	100 mW, 20 dBm	ca. 100 m
Klasse 2	2,5 mW, 4 dBm	ca. 10 m
Klasse 3	1 mW, 0 dBm	ca. 1 m

**Tabelle 10.1.** Sendeleistung und Übertragungreichweite für verschiedene Bluetooth-Klassen.

Um die Robustheit der Übertragung zu erhöhen, werden bei Bluetooth FEC<sup>9</sup> und ARQ<sup>10</sup> für die Fehlerkorrektur verwendet. Auch wenn die Robustheit damit erheblich verbessert wird, so kann keine Garantie für vollständig korrekte Daten gegeben werden. Bei stark verrauschter Umgebung oder bei sicherheitskritischen Anwendungen sollten daher weitere Korrekturmechanismen implementiert werden.

Ein potenzielles Sicherheitsrisiko besteht beim *Pairing*<sup>11</sup> der Geräte, weil dafür eine PIN<sup>12</sup> übertragen werden muss. Diese könnte von einer dritten Stelle abgehört und entschlüsselt werden, die dann Zugriff auf das Gerät erlangen würde. Es empfiehlt sich deshalb, die bei einer Authentifizierung erzeugten Verbindungsschlüssel (*Link Keys*) lokal permanent zu speichern und anschließend die Möglichkeit zur Neuauthentifizierung abzuschalten. Ein erneuter Verbindungsaufbau erfolgt dann auf Basis der hinterlegten Verbindungsschlüssel und eine wiederholte Pin-Authentifizierung wird vermieden. Eine möglichst lange PIN verbessert die Sicherheit zusätzlich, da die Entschlüsselung erschwert wird. Im Bluetooth-Protokoll sind für eine PIN bis zu 16 beliebige Zeichen vorgesehen. Die in der Praxis mögliche Anzahl hängt jedoch auch davon ab, wie viele Stellen der Treiber unterstützt.

Die Übertragungssicherheit wird neben der Authentifizierung durch eine Verschlüsselung der Daten zusätzlich erhöht. Grundlage hierfür ist ebenfalls der beim Pairing erzeugte Verbindungsschlüssel. Für die Verschlüsselung eines Datenpaketes wird dazu bei Aktivierung der Verschlüsselung zwischen zwei Geräten aus dem Verbindungsschlüssel ein neuer Schlüssel erzeugt. Die Länge richtet sich nach dem kürzeren der beiden Verbindungsschlüssel. Als Verschlüsselungsalgorithmus kommt die sog. Stromverschlüsselung<sup>13</sup> zum Einsatz, bei welcher die Bits des Datenstroms mit einem Schlüsselstrom XOR-verknüpft werden. Die Verschlüsselung ist eine optionale Sicherheitsfunktion und nicht automatisch aktiviert.

Für die folgenden Experimente wird ein USB-Bluetooth-Stick benötigt, wobei die Klasse frei wählbar ist und nur vom gewünschten Einsatzgebiet abhängt.

#### 10.4.2 Die Werkzeuge *Bluez-Utils*

Bluez ist der offizielle *Linux Bluetooth Stack* [Bluez 08] und wird von allen gängigen Linux-Distributionen unterstützt. Bluez implementiert dazu meh-

<sup>9</sup> Engl. *Forward Error Correction* (Vorwärtsfehlerkorrektur). Kodiert Daten redundant, d. h. der Empfänger kann diese ohne Rückfrage selbst korrigieren.

<sup>10</sup> Engl. *Automatic Repeat Request*. Mechanismus um fehlerhafte Pakete erneut anzufordern.

<sup>11</sup> Bezeichnung für den Zusammenschluss von Bluetooth-Geräten, bei dem eine Authentifizierung stattfindet.

<sup>12</sup> Persönliche Identifikationsnummer.

<sup>13</sup> Engl. *Stream Cipher*.

rere Protokolle auf verschiedenen Schichten des ISO/OSI-Modells und bringt neben Kernelmodulen, Gerätetreibern und Bibliotheken auch Werkzeuge zur Administration und Anwendung mit.

Der Bluez-Stack unterstützt folgende Bluetooth-Profile:

- Generic Access Profile – Zugriffssteuerung und Authentifizierung
- Service Discovery Profile – Auffinden von Diensten
- Serial Port Profile – Serielle Datenübertragung
- Dial-Up Networking Profile – Einwahlverbindung
- LAN Access Profile – PPP-Netzwerkverbindung
- OBEX Object Push Profile – Übertragung von Terminen und Adressen
- OBEX File Transfer Profile – Übertragung von Dateien
- PAN Profile – Netzwerkverbindung

Für die Installation der Werkzeugsammlung steht für Debian das Paket **bluez-utils** zur Verfügung, die notwendigen Kernelmodule sind in der Version 2.6 bereits enthalten. Neben dem gleichnamigen **ipkg**-Paket ist unter OpenWrt zusätzlich das Kernelpaket **kmod-bluetooth** zu installieren. Für Puppy existieren die inoffiziellen **.pet**-Pakete **BlueZ-libs-3.24.pet** und **BlueZ-utils-3.24.pet**.<sup>14</sup> Im vorliegenden Abschnitt werden lediglich die Kommandozeilenwerkzeuge vorgestellt und verwendet. Wer unter Debian direkt auf dem Bluetooth-Stack entwickeln möchte, muss zusätzlich die Bluez-Bibliothek als **libbluetooth-dev** installieren.

Nach der Installation der Pakete sollte zunächst ein erster Versuch unternommen werden, den eingesteckten Bluetooth-Dongle zu erkennen.<sup>15</sup> Folgender Aufruf (mit Root-Rechten) zeigt die Daten des Adapters an:

```
$ hciconfig
hci0: Type: USB
      BD Address: 00:11:22:33:44:55 ACL MTU: 310:10 SCO MTU: 64:8
      UP RUNNING PSCAN ISCAN
      RX bytes:2223 acl:0 sco:0 events:59 errors:0
      TX bytes:379 acl:0 sco:0 commands:27 errors:0
```

Mit der Option **-a** können erweiterte Einstellungen abgefragt und über **hciconfig hci0 features** die Fähigkeiten des Adapters ermittelt werden. Unter OpenWrt und Puppy ist der Daemon vorher noch von Hand mittels **hcid** zu starten. Für einen automatischen Start kann unter Debian in **/etc/default/bluetooth** folgender Parameter eingetragen werden:

<sup>14</sup> Diese sind unter **<embedded-linux-dir>/src/puppy/** verfügbar oder können von <http://www.murga-linux.com/puppy/viewtopic.php?t=25009> heruntergeladen werden.

<sup>15</sup> Für Puppy müssen zunächst die folgenden Kernelmodule mit dem Befehl **modprobe** geladen werden: **bluetooth**, **l2cap**, **rfcomm** und **hci-usb**.

```
BLUETOOTH_ENABLED=1
```

Unter OpenWrt bzw. Puppy können hierzu Startskripte in `/etc/init.d/` angelegt werden. In der Konfigurationsdatei `/etc/bluetooth/hcid.conf` sind die zentralen Bluetooth-Einstellungen hinterlegt. Im Abschnitt `options` sollten folgende Einträge enthalten sein:

```
options {
    # Automatically initialize new devices
    autoinit yes;

    # Security Manager mode
    security auto;

    # Pairing mode
    pairing multi;

    # Default PIN code for incoming connections
    passkey "your-pin";
}
```

Für den Adapter selbst können in der Rubrik `device` noch weitere Anpassungen vorgenommen werden. Hier wird u. a. auch der Name angegeben, unter welchem der Adapter von anderen Geräten gefunden wird. Die Klasse `0x000100` gibt an, dass es sich um ein Bluetooth-Gerät von Typ Computer handelt. Prinzipiell kann sich der Stick mit einer anderen Geräteklasse z. B. auch als Mobiltelefon ausgeben.<sup>16</sup> Die Geräteeinstellungen könnten wie folgt aussehen:

```
device {
    # Local device name
    name "my-bt-adapter";

    # Local device class
    class 0x000100;

    # Default packet type
    #pkt_type DH1,DM1,HV1;

    # Inquiry and Page scan
    iscan enable; pscan enable;
    discovto 0;

    # Default link mode
    lm accept;

    # Default link policy
    lp rswitch,hold,sniff,park;

    # Authentication and Encryption (Security Mode 3)
    auth enable;
    #encrypt enable;
}
```

Der Stick sollte zumindest zu Beginn der Versuchsphase so eingestellt sein, dass er eingehende Verbindungen stets akzeptiert. Wichtig ist ebenfalls der Eintrag

<sup>16</sup> Die Bluetooth-Geräteklassen sind in der Spezifikation V2.1 unter <http://www.bluetooth.com/Bluetooth/Technology/Building/Specifications/> aufgeführt.

`discovto 0;`, der nicht immer gesetzt ist und der den Stick erst nach außen sichtbar macht.<sup>17</sup> Um eine Verschlüsselung verwenden zu können, muss diese von beiden Partnern unterstützt werden, sonst schlägt die Verbindung fehl. Beim Testen sollten mögliche Fehlerquellen ausgeschlossen werden, sodass die Verschlüsselung zunächst besser abgeschaltet bleibt. Nach der Versuchsphase ist eine Verschlüsselung aber in jedem Falle zu empfehlen.

Der Bluetooth-Daemon wird nun neu gestartet:<sup>18</sup>

```
$ sudo /etc/init.d/bluetooth restart
```

Mit einem Mobiltelefon oder einem anderem Bluetooth-Gerät kann nun die Suche nach dem Stick initiiert werden. Wenn der Adapter in die Geräteliste aufgenommen werden soll, so ist auf Anfrage die in `/etc/bluetooth/hcid.conf` hinterlegte Pin einzugeben. Für Puppy ist vor einer ersten Verbindungsaufnahme der Passkey-Daemon mit dem Befehl `passkey-agent --default <your-pin> &` zu starten. Nachdem eine Verbindung aufgebaut wurde, wird der `passkey-agent` nicht mehr benötigt.

Sobald das Pairing erfolgreich abgeschlossen wurde, wird in der Datei `/var/lib/bluetooth/<bt-mac-address>/linkkey` ein neuer Eintrag mit dem Schlüssel des gepaarten Bluetooth-Gerätes angelegt. In diesem Verzeichnis werden sämtliche Informationen der zugeordneten Geräte abgelegt. Verwehrt sich ein Gerät nach mehreren Konfigurationsversuchen einer Anmeldung, so kann es hilfreich sein, dieses Verzeichnis komplett zu löschen.

Das Programm `hcitool` ermöglicht den Test weiterer Bluetooth-Features. Der folgende Befehl sucht nach sichtbaren Bluetooth-Geräten und gibt deren MAC-Adresse aus:

```
$ hcitool scan
Scanning ...
11:22:33:11:22:33 Joa's Phone
00:1E:45:05:00:00 K800i
00:19:B7:7F:00:00 Ck
00:1E:C2:B5:00:00 i61mac13
```

Mit der Option `inq` liefert das Programm zusätzlich den Zeitunterschied und den Klassentyp:

```
$ hcitool inq
Inquiring ...
00:1E:C2:B5:00:00 clock offset: 0x030a class: 0x30210c
00:1E:45:05:00:00 clock offset: 0x595b class: 0x5a0204
11:22:33:11:22:33 clock offset: 0x2f92 class: 0x520204
00:19:B7:7F:00:00 clock offset: 0x7ca2 class: 0x5a0204
```

<sup>17</sup> Wenn der Bluetooth-Stick dennoch nicht gefunden wird, so ist der Eintrag in `/var/lib/bluetooth/<bt-mac-address>/config` zu überprüfen. Unter Umständen muss hier `mode connectable` in `mode discoverable` geändert werden. Aus unerklärlichen Gründen geschieht dies bei einem Nachtrag von `discovto 0;` nicht immer automatisch.

<sup>18</sup> Unter Puppy wird die Prozess-ID mit `ps | grep hcid` gefunden, der Daemon dann mittels `kill -9 <pid>` beendet und anschließend neu gestartet.

Direkte Verbindungen zu Geräten lassen sich über folgenden Befehl aufbauen:

```
$ sudo hcitool cc 11:22:33:11:22:33
```

Anschließend kann mit der Option `auth` eine Authentifizierung angestoßen werden. Der Parameter `dc` schließt die Verbindung wieder. Falls beim Verbindungsaufbau Probleme auftreten, so kann dies an der Rolle der Geräte liegen, welche über die Option `--role=m,s` (immer Master bzw. Rollenwechsel zulässig) vorgegeben wird. Link-Modus und -Policy in `/etc/bluetooth/hcid.conf` spezifizieren das grundlegende Rollenverhalten. Die aktuell aktiven Verbindungen werden über folgenden Befehl aufgelistet:

```
$ hcitool con
Connections:
< ACL 11:22:33:11:22:33 handle 43 state 1 lm MASTER
```

Andere Geräte lassen sich übrigens auch mit dem Befehl `l2ping`, analog zum Netzwerkbefehl `ping`, ansprechen und reagieren auf eine Echoanforderung:

```
$ sudo l2ping 11:22:33:11:22:33
Ping: 11:22:33:11:22:33 from 00:11:22:33:44:55 (data size 44) ...
44 bytes from 11:22:33:11:22:33 id 0 time 18.87ms
44 bytes from 11:22:33:11:22:33 id 1 time 67.94ms
44 bytes from 11:22:33:11:22:33 id 2 time 70.97ms
3 sent, 3 received, 0% loss
```

Mit dem Programm `sdptool` können Dienste eines anderen Gerätes über das *Service Discovery Protocol* abgefragt werden. Dies liefert auch Informationen, ob bspw. ein Datentausch über OBEX möglich ist oder ob das Gerät als Fernsteuerung verwendet werden kann (vgl. Abschnitte 10.4.3 und 10.4.5). Für eine Ausgabe der Liste verfügbarer Dienste ist folgender Befehl auszuführen:

```
$ sdptool browse 11:22:33:11:22:33
```

Als Ausgabe folgt bei Mobiltelefonen üblicherweise eine lange Liste von Protokollen und deren Beschreibungen. Für ein Sony Ericsson W810 erhält man folgende Ausgaben (gekürzt):

```
$ sdptool browse 11:22:33:11:22:33
Browsing 11:22:33:11:22:33 ...
Service Description: Sony Ericsson W810
Service RecHandle: 0x10000
Service Class ID List:
"PnP Information" (0x1200)
...
Service Name: OBEX SyncML Client
Service RecHandle: 0x10001
...
Service Name: Dial-up Networking
Service RecHandle: 0x10002
...
Service Name: Serial Port
Service RecHandle: 0x10003
...
Service Name: HS Voice Gateway
Service RecHandle: 0x10005
...
Service Name: OBEX File Transfer
```

```

Service RecHandle: 0x10007
...
Service Name: OBEX IrMC Sync Server
Service RecHandle: 0x10008
...
Service Name: Mouse & Keyboard
Service Description: Remote Control
Service Provider: Sony Ericsson
Service RecHandle: 0x1000a
Service Class ID List:
    "Human Interface Device" (0x1124)
...

```

Die OBEX-Dienste ermöglichen das Senden von Dateien (*OBEX File Transfer*) oder eine Synchronisation der Daten (*OBEX Sync Service*). Das *Dialup Networking Profile* kommt bei einer Nutzung als Modem zum Aufbau einer PPP-Verbindung zum Einsatz. Über das *Serial Port Profile* können Daten wie über eine serielle Verbindung übertragen werden. Bluetooth dient hierbei sozusagen als Funkbrücke.

Audio-Dienste erlauben die Nutzung einer Freisprecheinrichtung oder eines Headsets, wobei im vorliegenden Falle der Dienst *HS Voice Gateway* diese Aufgabe übernimmt. Der Dienst mit Namen *Mouse & Keyboard* erlaubt eine Nutzung des Mobiltelefons als HID-Eingabegerät.<sup>19</sup> Dies wiederum ermöglicht eine leichte Integration in das System, da für Standardgeräte dieses Typs keine speziellen Treiber notwendig sind. In Abschnitt 10.4.5 wird eine Anwendung vorgestellt, die auf diesem Dienst basiert.

### 10.4.3 Datentransfer mit ObexFTP

Mit dem Programm `obexftp`<sup>20</sup> lassen sich Dateien zwischen Bluetooth-Geräten austauschen. ObexFTP setzt auf dem *OBEX File Transfer Protocol* auf und eignet sich sehr gut zur Verwendung in eigenen Skripten als individuelles Werkzeug für Synchronisationsaufgaben. Mit folgender Befehlszeile kann eine Textdatei an ein Bluetooth-Gerät gesendet werden:

```

$ echo "greetings from host" > test.txt
$ obexftp -b 11:22:33:11:22:33 -p test.txt
Browsing 11:22:33:11:22:33 ...
Channel: 7
Connecting...done
Sending "test.txt"... done
Disconnecting...done

```

Ein Mobiltelefon wird diese Datei in einem bestimmten lokalen Verzeichnis ablegen, wobei der Pfad abhängig vom Dateityp ist. Da sich `obexftp` auch über

<sup>19</sup> Engl. *Human Interface Device*, vgl. auch Abschnitt 7.5.

<sup>20</sup> Enthalten im gleichnamigen Debianpaket. Für Puppy sind die `.pet`-Dateien `Obexftp-0.20` und `Openobex-1.3-apps` zu installieren. Diese sind unter `<embedded-linux-dir>/src/puppy/` oder <http://www.murga-linux.com/puppy/viewtopic.php?t=25009> verfügbar.

USB-, IrDA- oder Netzwerkverbindungen nutzen lässt, muss ein Bluetooth-Partner explizit mit **-b** angegeben werden. Mit der Option **-l** kann eine Verzeichnisliste des entfernten Gerätes im XML-Format ausgegeben werden:

```
$ sudo obexftp -b 11:22:33:11:22:33 -l /Telefonspeicher
Browsing 11:22:33:11:22:33 ...
Channel: 7
Connecting...done
Receiving "/Telefonspeicher"... Sending "... done
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE folder-listing SYSTEM "obex-folder-listing.dtd">
<!--
  XML Coder, May 24 2006, 17:46:10, (C) 2001 Sony Ericsson Mobile
  Communications AB
-->
<folder-listing version="1.0"><folder name="Designs"/>
<folder name="Videos"/>
<folder name="Andere"/>
<folder name="Bilder"/>
<folder name="Sounds"/>
</folder-listing>
done
Disconnecting...done
```

Für einen Datentransfer vom anderen Gerät kann mit dem Parameter **-c** in ein Verzeichnis gewechselt werden. Der Kopiervorgang wird durch den zusätzlichen Parameter **-g** eingeleitet:

```
$ sudo obexftp -b 11:22:33:11:22:33 -c /Memory\ Stick/DCIM/100MSDCF/ -g
DSC00246.JPG DSC00247.JPG
Browsing 11:22:33:11:22:33 ...
Channel: 7
Connecting...done
Sending "... Sending "Memory Stick"... Sending "DCIM"... Sending "100MSDCF
"... done
Receiving "DSC00246.JPG"...done
Receiving "DSC00247.JPG"...done
Disconnecting...done
```

Die XML-Antwortdaten können bspw. mit dem schlanken XML-Parser *TinyXML* weiterverarbeitet werden.<sup>21</sup> Mit diesen beiden Werkzeugen steht der Entwicklung von Anwendungen oder Skripten zur Adresssynchronisation sowie dem Datenabgleich von Bildern und SMS-Nachrichten nichts mehr im Wege.

#### 10.4.4 Serielle Bluetooth-Kommunikation und AT-Befehle

Über das Programm **rfcomm** kann eine Bluetooth-Verbindung wie eine serielle Schnittstelle betrieben werden. Genauer gesagt handelt es sich bei *RFCOMM* um ein Bluetooth-Protokoll, welches zur Emulation serieller Schnittstellen genutzt wird. Das *Serial Port Profile* definiert die dazu notwendigen Anforderungen. Dieser Dienst wird verwendet, um Mobiltelefone als Modem zu betreiben. Er wird aber auch von vielen anderen Bluetooth-Geräten zur seriellen Datenübertragung genutzt (vgl. BTM222-Modul am Ende dieses Abschnitts).

<sup>21</sup> Verfügbar unter <http://www.grinninglizard.com/tinyxml/>.



Ein großer Vorteil dieses Protokolls ist die Möglichkeit, mit wenig Aufwand Anwendungen der seriellen Kommunikation auf Bluetooth-Geräte portieren zu können.

In der Datei `/etc/bluetooth/rfcomm.conf` können RFCOMM-Geräte definiert werden, die für eine Kommunikation mit bestimmten Bluetooth-Geräten zuständig sind. Diese Geräte werden dann bereits beim Systemstart angelegt.

Mit folgendem Eintrag wird eine Gerätedatei `/dev/rfcomm0` erstellt und dieser ein Mobiltelefon mit bekannter MAC-Adresse zugeordnet:

```
rfcomm0 {
    bind yes;
    # MAC address of mobile phone:
    device 11:22:33:11:22:33;
}
```

Nach einem Neustart des Bluetooth-Daemons können über die Gerätedatei Daten an das Mobiltelefon gesendet werden:

```
$ sudo /etc/init.d/bluetooth restart
echo 1 > /dev/rfcomm0
```

Der Verbindungsstatus aller Einträge kann durch einen Aufruf ohne Zusatzparameter erfragt werden:

```
$ rfcomm
rfcomm0: 11:22:33:11:22:33 channel 1 connected [tty-attached]
rfcomm1: 11:11:11:22:22:22 channel 1 clean
```

Zur Steuerung von PC-Modems wurde in den 80er Jahren ein Satz sog. AT-Kommandos spezifiziert, der auch als Hayes-Befehlssatz bekannt ist. Dieser Befehlssatz wurde in weiten Teilen für Mobiltelefone übernommen und erweitert, sodass sich nach wie vor Wählvorgänge und individuelle Statusabfragen darüber ausführen lassen. Wird bspw. über das Programm `minicom` oder das *Serial Port Terminal* der Befehl `ATI`<sup>22</sup> auf die in `/etc/bluetooth/rfcomm.conf` konfigurierte Schnittstelle abgesetzt, so erhält man folgende Antwort:

```
042
OK
```

Als weiteres Beispiel liefert der Befehl `AT+CBC` den aktuellen Zustand der Stromversorgung des Akkus zurück:

```
+CBC: 0,64
OK
```

<sup>22</sup> Jeder AT-Befehl beginnt mit den Zeichen `AT`, dann folgt das Kommandokürzel. In diesem Fall werden mit `I` Informationen angefordert.

Die erste Nummer gibt an, ob der Ladeadapter angeschlossen ist, danach folgt der Ladezustand in Prozent. Für einen Wählvorgang wird die Syntax `ATD<nummer>;` verwendet (D für *Dial*).

Die implementierten AT-Befehle hängen vom Typ des Telefons ab und können auf den Herstellerseiten nachgelesen werden.<sup>23</sup> Für die Einrichtung einer Internetverbindung mithilfe eines Mobiltelefons sei an dieser Stelle auf die Website von Tom Yates verwiesen.<sup>24</sup>

Die Fa. Rayson bietet günstige Bluetooth-Module mit UART-Schnittstelle an ([Rayson 08], vgl. Bezugsquellen in Tabelle E.1). Diese eignen sich zum Aufbau serieller Funkstrecken und können Mikrocontroller oder andere Geräte mit serieller Schnittstelle mit einer Bluetooth-Verbindung ausstatten. Die Module werden zunächst über die UART-Schnittstelle mit AT-Befehlen konfiguriert und wechseln bei einer bestehenden Bluetooth-Verbindung automatisch vom Kommando- in den Datenübertragungsmodus.

#### 10.4.5 Das Mobiltelefon als Fernbedienung

Viele Mobiltelefone des Herstellers *Sony-Ericsson* besitzen die Fähigkeit, sich über Bluetooth als HID-Eingabegerät zur Verfügung zu stellen. Ob dies für das eigene Telefon gilt, kann über `sdptool browse <bt-mac>` geprüft werden. Bluetooth-Tastaturen oder -Mäuse basieren auf dem gleichen Protokoll und werden von fast allen Linux-Systemen unterstützt. Die Einbindung eines HID-Mobiltelefons kann entsprechend ebenfalls ohne weitere Treiber erfolgen.

Ist eine Verwendung des Mobiltelefons nicht möglich, so können alternativ andere HID-Eingabegeräte verwendet werden (vgl. Tabelle E.1). Für eine HID-Unterstützung müssen die Kernelmodule `hid`, `usbhid` und `evdev` installiert sein. Während dies bei Desktop-Distributionen meist der Fall ist, so sind auf manchen Distributionen für Embedded-Systeme (z. B. OpenRISC Alekto) diese Eingabegeräte nicht standardmäßig vorgesehen. Im Falle des Alekto müssen diese Module in den Kernel integriert und dieser dazu neu gebaut werden (zumindest gilt dies für `evdev`, vgl. auch Abschnitt 5.7). Unter OpenWrt ist zusätzlich das Paket `kmod-usb-hid` zu installieren.

Der ebenfalls in Bluez-Utils enthaltene Bluetooth-HID-Daemon `hidd` baut mit folgendem Befehl eine Verbindung zu einem HID-Gerät auf:

```
$ sudo hidd --connect 11:22:33:11:22:33
```

Wurde zuvor bereits ein Pairing durchgeführt, so erfolgt der Verbindungsaufbau ohne Nachfrage. Auf dem Mobiltelefon wird die HID-Anfrage entgegengenommen und üblicherweise nachgefragt, ob die Fernbedienungsfunktion gestartet werden soll. Wird dies bestätigt und ein Profil ausgewählt, so bleibt die

<sup>23</sup> Für Sony-Telefone auf der Website der Sony Developer World [SEDeveloper 08].

<sup>24</sup> <http://www.teaparty.net/technotes/blue-gprs.html>.

Verbindung erhalten. Der Verbindungsstatus kann jederzeit wie folgt überprüft werden:

```
$ hidd
11:22:33:11:22:33 Sony Ericsson Remote Control [0000:c042] connected
```

Für das Anlegen der Fernsteuerungsprofile steht für die SE-Mobiltelefone das Programm *Sony Ericsson Bluetooth Remote Control* zur Verfügung.<sup>25</sup>

Mit diesem werden die Tasten des Handys den Funktionen der Maus oder der Computertastatur zugeordnet und aus dieser Zuordnung ein Fernsteuerungsprofil erstellt. Nachdem das Profil per Kabel oder Bluetooth auf das Mobiltelefon übertragen wurde, kann dieses als Computertastatur mit eigener Belegung eingesetzt werden.



**Abb. 10.1.** Datenübertragung mit `obexftp` und Fernsteuerungsfunktion des Sony W810i.

Wurde die Fernsteuerungsfunktion von einem Ubuntu-Rechner aus aufgerufen, dann lassen sich je nach Profil mit dem Handy Maus oder Tastatur simulieren. Wenn auf dem PC eine Konsole geöffnet ist, so können die Tasten von einer Konsolenanwendung einfach über `stdin` eingelesen werden.

Eine ähnliche Vorgehensweise ist auch bei eingebetteten Systemen möglich. Jedoch erfolgt die Anmeldung üblicherweise über SSH, womit die Eingaben einer am Gerät angeschlossenen Tastatur natürlich nicht im SSH-Terminal zu sehen sind. Eine Weiterleitung auf das SSH-Terminal ist keine verlässliche

<sup>25</sup> Für MS Windows oder MAC OS X unter [SEDeveloper 08] verfügbar.

Lösung: Wird eine Linux-Anwendung beim Hochfahren automatisch gestartet, so existiert keine Shell, auf welche die Tastatureingaben weitergeleitet werden könnten.

Eine mögliche Lösung scheint aber der Abgriff der Tastatureingaben direkt vom angeschlossenen Gerät bzw. von der Event-Schnittstelle, die Teil des Input-Subsystems von Linux ist. Da gerade Benutzereingaben sehr unregelmäßig eintreffen, werden im System sog. Events ausgelöst, auf die bestimmte Programme reagieren können. Als Event-Schnittstellen existieren virtuelle Geräte in `/dev/input/` mit Namen `event<x>`, die einem bestimmten Eingabegerät zugeordnet sind.

Welchen Event-Handler das System für die angeschlossene Fernsteuerung verwendet, kann über nachfolgenden Befehl herausgefunden werden. Eine aktive `hidd`-Verbindung wird hierbei vorausgesetzt:

```
$ cat /proc/bus/input/devices
I: Bus=0005 Vendor=0000 Product=c042 Version=0000
N: Name="Sony Ericsson Remote Control"
P: Phys=00:11:67:78:DF:02
S: Sysfs=/devices/pci0000:00/0000:00:1d.3/usb4/4-1/4-1:1.0/hci0/
  ac10018130F2A81/input/input8
U: Uniq=11:22:33:11:22:33
H: Handlers=kbd mouse2 event6
B: EV=120017
B: KEY=70000 0 0 0 0 e080ffdf 1cffff ffffffff fffffffe
B: REL=3
B: MSC=10
B: LED=1f
```

In der Liste der Einträge taucht ein Abschnitt auf, der auf das Mobiltelefon hinweist. In der Auflistung der Handler (Zeile `H:`) sollte ein Event-Eintrag `event<x>` enthalten sein. Ist dies nicht der Fall, so besteht ein Problem mit dem Event-Handler. In der Regel ist das Modul `evdev` dann nicht geladen. Wird die Gerätedatei direkt ausgelesen, so erscheinen beim Betätigen der Tasten am Mobiltelefon kryptische Zeichen in der Konsole:

```
$ cat /dev/input/event<x>
```

Der Grund hierfür ist, dass nicht die einzelnen Zeichen der Tastatur gesen-det werden, sondern jeweils eine Struktur vom Typ `input_event`. Diese ist aufgebaut wie folgt:

```
#include <linux/input.h>

struct input_event {
    struct timeval time;
    unsigned short type;
    unsigned short code;
    unsigned int   value;
};
```

Neben der Zeit `time` und dem Typ des Events `type` (Wert `EV_KEY` für Tastatureingaben) ist der Tastencode in `code` enthalten. Weiterhin gibt `value` an, ob die Taste gedrückt (1) oder losgelassen wurde (0). Die Auswertung der

Struktur sollte in einer Anwendung erfolgen. Das Beispiel `hid_input_event` zeigt hierzu die Vorgehensweise bei der Tastenidentifizierung.

Mit diesem Hintergrundwissen kann das Mobiltelefon nun in einer realistischen Anwendung eingesetzt werden.

Als Beispiel wird die in Kapitel 7 vorgestellte serielle Relaiskarte über Bluetooth angesteuert (`bluetooth_remote`). Die Argumente sind das Event-Eingabegerät und die Adresse der seriellen Schnittstelle der Relaiskarte. Zu beachten sind weiterhin die Zugriffsrechte der einzelnen Gerätedateien. Die Ausführung mit `sudo` umgeht das Problem.

Wichtig ist, dass ein Fernsteuerungsprofil erstellt wurde, in welchem die Handy-Tasten eins bis acht auf die entsprechenden Computertasten gelegt wurden und dem Zeichen „#“ ein „e“ zugeordnet wurde. Letztere Zuordnung wird zum Beenden des Programms verwendet. Mit dem Startskript `start.sh` im gleichen Verzeichnis lassen sich alle notwendigen Schritte (Verbindung aufbauen – Programmstart – Verbindung schließen) nacheinander ausführen – wieder sind Root-Rechte erforderlich. Die Bluetooth-Gerätenummer und die Systempfade sind zuvor im Skript anzupassen:

```
$ ./start.sh
```

Bei einem Start mit dem Befehl `nohup` wird die Anwendung im Hintergrund ausgeführt und bleibt auch dann aktiv, wenn die SSH-Verbindung beendet wird:

```
$ nohup ./start.sh &
```

## 10.5 USB-GPS-Module

Für einen mobilen Einsatz des eingebetteten Systems kann es sinnvoll sein, über GPS die Position zu erkennen bzw. die zurückgelegte Strecke zu protokollieren. Auch wenn eigene GPS-Lösungen kaum an den Funktionsumfang kommerzieller Geräte heranreichen können, so bietet eine offene Schnittstelle zu einem GPS-Empfänger zumindest den Vorteil der einfachen Integration in eigene Anwendungen.

Im folgenden Abschnitt wird der USB-GPS-Empfänger NL-302U von Navilock verwendet, der sich vergleichsweise einfach über das verbreitete NMEA-0183-Protokoll<sup>26</sup> auslesen lässt (vgl. Tabelle E.1). Weiterhin existiert für den Umgang mit dem Gerät bereits eine Reihe von Werkzeugen.

<sup>26</sup> Kommunikationsstandard der *National Marine Electronics Association (NMEA)*, vgl. <http://www.nmea.org/>.

Der NL-302U besitzt einen eingebauten USB-Seriell-Wandler auf der Basis des PL2303 und meldet sich beim Anstecken an ein Debian-System entsprechend unter `/dev/ttyUSB0` an.<sup>27</sup>

Unter OpenWrt wird das Kernelpaket `kmod-usb-serial-pl2303` benötigt. Unter Puppy wird das entsprechende Modul mit dem Befehl `modprobe pl2303` geladen – der serielle Anschluss steht dort dann unter `/dev/usb/ttyUSB0` bereit.

### 10.5.1 Der GPS-Daemon GPSD

Das GPSD-Projekt [GPSD 08] stellt zahlreiche Werkzeuge zur Kommunikation mit GPS-Empfängerbausteinen zur Verfügung. Dazu zählen der GPS-Daemon `gpsd` und Client-Testprogramme wie `xgps` oder `cgps`. Für Debian können Daemon und Clientprogramme über die Pakete `gpsd` und `gpsd-clients` installiert werden, für OpenWrt ist der Daemon im Paket `gpsd` verfügbar. Die Kommunikation zwischen Daemon und Clients erfolgt über eine TCP-Socketverbindung. Wird zunächst der Empfänger am Embedded-System angesteckt, so ist die Baudrate auf die im NMEA-Standard definierten 4800 Baud einzustellen:

```
$ stty -F /dev/ttyUSB0 speed 4800
```

Der Daemon kann nun über folgenden Befehl mit Debug-Ausgaben (`-D2`) im Vordergrund (`-N` gestartet werden):

```
$ gpsd -D2 -n -N /dev/ttyUSB0
```

Hierbei bewirkt die Option `-n`, dass `gpsd` auch ohne Client-Verbindung den Adapter fortlaufend ausliest. Standardmäßig erfolgt die Kommunikation des `gpsd` mit Client-Programmen über den Port 2947. Spätestens ca. 40 Sekunden nach dem Einstecken sollten Positionsdaten in der Ausgabe sichtbar werden (die Wartezeit ist erforderlich für den ersten Verbindungsaufbau zwischen dem Empfänger und den GPS-Satelliten):

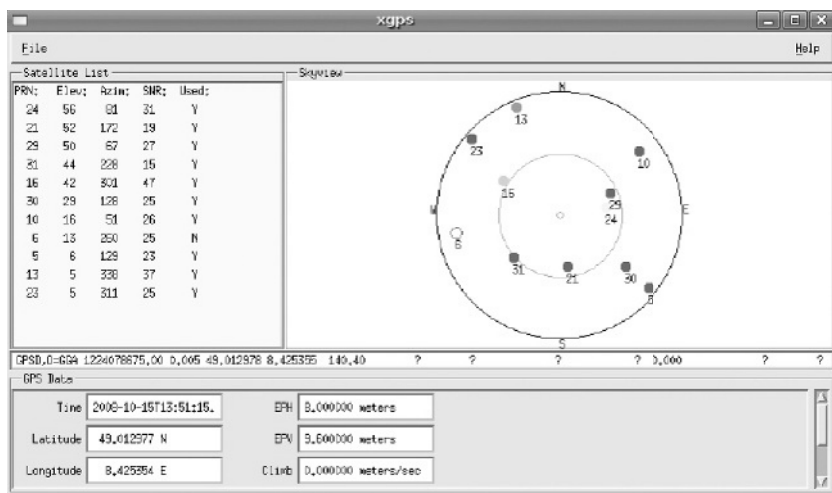
```
...
gpsd: <= GPS: $GPVTG,236.35,T,,M,0.06,N,0.1,K*66
gpsd: <= GPS: $GPGGA,155835.000,4900.7757,N,00825.5268,E,1,10,1.4,149.0,M
,47.9,M,,0000*58
...
```

Voraussetzung hierfür ist eine Sichtverbindung zwischen Empfänger und Satelliten. Die Daten können nun mit dem Client-Programm `cgps` unter Angabe der Hostadresse vom Daemon über den dort eingestellten Port bezogen werden:

<sup>27</sup> Für den OpenRISC Alekto muss der PL2303-Treiber in den Kernel integriert werden. Dies geschieht in der Kernelkonfiguration unter *Device Drivers/USB Support/USB Serial Converter Support* (vgl. auch Abschnitt 5.7).

```
$ cgps 192.168.1.10:2947
```

Besteht eine Netzwerkverbindung zu einem Ubuntu-Rechner mit grafischer Oberfläche, so lässt sich dort zur Visualisierung das Programm **xgps** starten (vgl. Abbildung 10.2).



**Abb. 10.2.** Darstellung der GPS-Daten mit dem Programm **xgps** auf einem Ubuntu-Client. Der GPS-Daemon **gpsd** läuft auf dem Embedded-System mit IP 192.168.1.10.

### 10.5.2 GPS in der Anwendung

Für die Weiterverarbeitung der GPS-Daten in einer eigenen Anwendung existieren zwei Möglichkeiten:

1. Der indirekte Zugriff auf die GPS-Daten über eine GPSD-Instanz unter Verwendung der Bibliothek **libgps**. Hierfür ist keine Socket-Verbindung notwendig.
2. Der direkte Zugriff auf den Adapter unter Verwendung der Bibliothek **libgpsd**, ähnlich der Funktionsweise des Daemons selbst.

Theoretisch ist es auch möglich, selbst eine Socketverbindung zu öffnen. Dies ist bei Verwendung der **libgps** aber nicht notwendig. Das Beispiel **gps** realisiert die Kommunikation über die zweite Variante. Die notwendige **gpsd**-Bibliothek wird bei der Ausführung von **make** automatisch heruntergeladen und übersetzt. Für eine einfache Handhabung wurde die Klasse *GPSReceiver* entwickelt, die alle **gpsd**-Aufrufe kapselt (vgl. den folgenden Abschnitt).

### 10.5.3 Die Klasse GPSReceiver

Die Klasse GPS-Receiver<sup>28</sup> speichert GPS-Daten und nimmt Umrechnungen der Lat/Lon-Darstellung<sup>29</sup> nach UTM<sup>30</sup> vor. Diese Darstellung hat den Vorteil, dass die Positionsangaben in der aussagekräftigeren Einheit *Meter* vorliegen und die Zahlen durch die relative Angabe zum Feldursprung eine Größe von 950 000 nicht übersteigen.

Die GPS-Daten werden intern in folgenden Membervariablen vorgehalten:

```
struct gps_device_t session; // gpsd session
struct gps_context_t context; // gpsd context
int used_satellites; // number of used satellites
int satellites; // number of available satellites
double latitude; // latitude coordinate [deg]
double longitude; // longitude coordinate [deg]
double altitude; // altitude [m]
double speed; // speed [km/h]
double time; // timestamp [s]
double utm_x; // UTM x coordinate [m]
double utm_y; // UTM y coordinate [m]
```

Als Benutzerschnittstelle stehen Funktionen zum Verbindungsaufbau, zur Anforderung neuer GPS-Daten und zur Abfrage einzelner Werte zur Verfügung:

**GPSReceiver(char\* dev);**

Erzeugt ein Objekt vom Typ **GPSReceiver**, initialisiert eine Verbindung mit dem an *dev* angeschlossenen GPS-Empfänger und aktiviert diese.

**int updateData();**

Liest GPS-Daten vom Empfänger. Die jeweiligen Datenfelder werden nur aktualisiert, wenn ein gültiger Messwert empfangen wurde. Als Rückgabewert dient eine Bitmaske die angibt, welche Daten sich geändert haben (vgl. Maskendefinition in `gps/gpsd-2.37/gps.h`).

**int printData();**

Schreibt alle im Objekt gesammelten GPS-Daten auf die Standardausgabe.

**<type> get<value>();**

Liefert *<value>* im jeweiligen Typ *<type>* der GPS-Variablen zurück. Falls noch kein gültiger Wert empfangen wurde, so wird *-1* zurückgegeben.

Unsere Versuche, die **gpsd**-Bibliothek mit einem Cross Compiler zu übersetzen, blieben leider erfolglos. Die Anwendung kann aber nativ auf Embedded-Systemen übersetzt werden.

<sup>28</sup> Zu finden in den Beispielen unter `usb/gps/GPSReceiver.h`.

<sup>29</sup> Positionsdarstellung in Form von Längen- und Breitengraden.

<sup>30</sup> *Universal Transverse Mercator*. Kartesische Positionsdarstellung durch konforme Abbildung des Erdellipsoids in die Ebene. Durch die Aufteilung der Längengrade in 60 und der Breitengrade in 20 Zonen wird eine hinreichend genaue Abbildung für die jeweiligen Felder erreicht (vgl. [http://geology.isu.edu/geostac/Field\\_Exercise/topomaps/utm.htm](http://geology.isu.edu/geostac/Field_Exercise/topomaps/utm.htm)).



Die GPS-Daten könnten bspw. in einer Logdatei abgespeichert und anschließend weiterverarbeitet werden. Ein XML-basiertes Datenformat für GPS-Wegpunkte und -strecken ist das **gpx**-Format.<sup>31</sup> Bei Verwendung dieses Formates lassen sich die Daten in Google Earth importieren und dort visualisieren.

## 10.6 USB-Speichererweiterung

Die Verwendung eines USB-Speichersticks stellt eine einfache Möglichkeit dar, ein eingebettetes System um zusätzlichen Festspeicher zu erweitern. Der Einsatz eines solchen zusätzlichen Festspeichermediums kann z. B. notwendig werden, wenn für ein Home-Verzeichnis mehr Speicherplatz als geplant erforderlich wird. Auch beim Einsatz von OpenWrt reicht der ohnehin sehr geringe interne Speicher oft nicht aus, um weitere Pakete zu installieren.

Die Wahl sollte hierbei nicht unbedingt auf den preiswertesten USB-Stick fallen. Lese- und Schreibraten von mindestens 10–15 MB/s sind in jedem Fall zu empfehlen. Im folgenden Abschnitt wird zunächst die grundlegende Vorgehensweise zur Laufwerkseinbindung beschrieben. Dann wird in Abschnitt 10.6.2 die Auslagerung des Home-Verzeichnisses auf einen USB-Stick erklärt.

### 10.6.1 Partitionierung und Einbindung eines USB-Sticks

Zunächst ist der USB-Stick zu partitionieren und mit dem gewünschten Dateiformat zu formatieren. Dies geschieht am einfachsten an einem Ubuntu-basierten PC unter Verwendung des Programms *Partition Editor*. Der Vorgang kann aber auch mit den Kommandozeilenwerkzeugen **fdisk** und **mkfs**<sup>32</sup> durchgeführt werden. In beiden Fällen muss bekannt sein, unter welcher Gerätedatei der USB-Stick im System angesprochen wird. Über einen **dmesg**-Aufruf unmittelbar nach dem Einstecken lässt sich dies leicht feststellen:

```
$ dmesg
...
usb-storage: device found at 6
usb-storage: waiting for device to settle before scanning
  Vendor:          Model: USB DISK 200X      Rev: PMAP
  Type:           Direct-Access              ANSI SCSI revision: 00
SCSI device sdc: 4030464 512-byte hdwr sectors (2064 MB)
sdc: Write Protect is off
sdc: Mode Sense: 23 00 00 00
sdc: assuming drive cache: write through
SCSI device sdc: 4030464 512-byte hdwr sectors (2064 MB)
...
```

<sup>31</sup> *GPS Exchange Format*, vgl. <http://www.topografix.com/gpx.asp>.

<sup>32</sup> Enthalten im Debian-Paket **util-linux**.

Die auf dem neuen Datenträger `/dev/sdc` verfügbaren Partitionen sind als `/dev/sdc1`, `/dev/sdc2` usw. geführt. Der Datenträger kann nun mittels `fdisk` folgendermaßen partitioniert werden:

```
$ sudo fdisk /dev/sdc
```

Zusammen mit der Hilfe im Programm über `(m)` sind die nächsten Schritte fast selbsterklärend. Um eine durchgängige `ext3`-Partition zu erhalten, werden zunächst mit der Option `(d)` vorhandene Partitionen gelöscht, eine neue primäre Partition erstellt `(n)` und die Default-Werte für die Zylinderangaben übernommen. Nach einer Überprüfung mit `(p)`<sup>33</sup> kann die Partitionstabelle mit `(w)` übernommen werden. Nähere Informationen zur Verwendung von `fdisk` finden sich im *Linux Documentation Project*.<sup>34</sup>

Nachdem eine gültige Partition `/dev/sdc1` erstellt wurde, kann nun mit dem Befehl `mkfs` ein Dateisystem formatiert werden. Über die Option `-t` wird das Dateiformat angegeben. Übliche Formate für Linux-Dateisysteme sind `ext2` oder `ext3`. Soll der USB-Stick hingegen nicht dauerhaft in das Linux-System eingebunden werden, sondern als Wechseldatenträger auch z.B. unter MS Windows oder Mac OS X Anwendung finden, so empfiehlt sich eine Formatierung mit FAT32 (Typ `vfat`). In diesem Beispiel wird der Datenträger als Typ `ext3` formatiert:

```
$ sudo mkfs -t ext3 /dev/sdb1
```

Sobald der Datenträger formatiert ist, können einzelne Partitionen mithilfe von `mount` in das System eingehängt (*gemountet*) werden. Zuvor muss ein Verzeichnis angelegt werden, unter welchem die Verzeichnisstruktur des Datenträgers eingebunden wird:

```
$ sudo mkdir /newmountpoint
$ sudo mount -t ext3 /dev/sdc1 /newmountpoint
```

Die Option `-t` gibt wiederum den Typ des Dateisystems an. Der Befehl `umount` wirft eingehängte Laufwerke aus, als Option dient die Gerätedatei selbst oder auch das gemountete Verzeichnis. Ein Aufruf von `mount` listet die momentan eingehängten Laufwerke auf:

```
$ mount
/dev/sda1 on / type ext3 (rw,errors=remount-ro)
...
/dev/sdb1 on /home type ext3 (rw)
/dev/sdc1 on /newmountpoint type ext3 (rw)
```

Für den Fall, dass ausgewählte Datenträger bereits beim Systemstart gemountet werden sollen, können in der Datei `/etc/fstab`<sup>35</sup> die notwendigen Einträge

<sup>33</sup> Die ID sollte 83 sein und ist gegebenenfalls mit der Option `(t)` anzupassen.

<sup>34</sup> Vgl. [http://tldp.org/HOWTO/Partition/fdisk\\_partitioning.html](http://tldp.org/HOWTO/Partition/fdisk_partitioning.html).

<sup>35</sup> Engl. *File System Table*.

angelegt werden. Die folgende Auflistung zeigt einen Standardeintrag für eine NSLU2 mit Debian:<sup>36</sup>

#	<file system>	<mount point>	<type>	<options>	<dump>	<pass>
proc		/proc	proc	defaults	0	0
/dev/sda1		/	ext3	defaults,errors=remount-ro	0	1
/dev/sda2		none	swap	sw	0	0
/dev/sda1		/media/usb0	auto	rw,user,noauto	0	0
/dev/sda2		/media/usb1	auto	rw,user,noauto	0	0

In den ersten beiden Spalten wird zunächst eine Gerätepartition **<file system>** einem Verzeichnis **<mount point>** zugeordnet, unter dem diese eingehängt werden soll. Nach Angabe des Dateisystemtyps im Feld **<type>** folgen die Zugriffsoptionen **<options>** (**rw**=read/write, **ro**=read only, **noauto**=wird nicht sofort gemountet).

Oftmals wird als Vorgabe **defaults** gewählt, was wiederum den Optionen **rw,suid,dev,exec,auto,nouser,async** entspricht. In der fünften Spalte kann angegeben werden, ob für dieses Verzeichnis eine Datensicherung mit dem Programm **dump** durchgeführt werden soll. Das Feld **<pass>** definiert die Reihenfolge einer Überprüfung des Dateisystems mit **fsck**. Für das Root-Dateisystem sollte dieses Feld mit 1 belegt werden, andere Laufwerke erhalten eine höhere Zahl oder 0, um die Überprüfung zu deaktivieren.

Mit folgendem **fstab**-Eintrag kann die Partition **/dev/sdc1** des Datenträgers **/dev/sdc** dauerhaft in das System eingebunden werden:

```
/dev/sdc1 /newmountpoint ext3 defaults 0 2
```

### 10.6.2 Auslagerung des Home-Verzeichnisses auf einen USB-Stick

Ein Verschieben des Home-Verzeichnisses auf einen USB-Stick hat mehrere Vorteile: Zum einen wird dadurch auf dem ursprünglichen Datenträger wieder Platz frei, zum anderen werden damit die Nutzerdaten unabhängig vom System abgelegt. Somit ist eine Neuinstallation oder ein Flashen neuer Firmware möglich, ohne dass die Daten der Benutzer betroffen sind. Um die Home-Verzeichnisse für alle Benutzer zu verschieben, wird zunächst das Verzeichnis mit den aktuellen Nutzerdaten umbenannt. Dann wird ein neues Home-Verzeichnis angelegt:

```
$ sudo mv /home /old_home
$ sudo mkdir /home
```

Nun kann der neue Datenträger in **/home** eingehängt werden:

```
$ sudo mount -t ext3 /dev/sdc1 /home
```

<sup>36</sup> Unter OpenWrt existiert standardmäßig keine Datei **/etc/fstab**. Diese kann jedoch nachträglich erstellt werden und wird dann automatisch berücksichtigt.

Für die vollständige Übertragung aller Hard- und Softlinks wird der Befehl `cpio` verwendet, da ein Kopiervorgang mit `cp` kein vollständiges Abbild liefern würde.<sup>37</sup> Folgende Befehle kopieren die Home-Verzeichnisse komplett:

```
$ cd /old_home
--ggf. mit su als root einloggen--
$ find . -depth -print0 | cpio --null --sparse -pvd /home
```

Durch einen entsprechenden Eintrag in `/etc/fstab` wird sichergestellt, dass der Datenträger bereits beim Systemstart in `/home` eingehängt wird:

```
/dev/sdc1 /home    ext3    defaults    0 2
```

Nach einem Neustart werden die Home-Verzeichnisse auf dem neuen Datenträger verwendet. Als Alternative zum Einhängen eines anderen Datenträgers in `/home` kann auch in der Datei `/etc/passwd` für jeden Benutzer die Stelle des Home-Verzeichnisses im Dateisystem angegeben werden:

```
#joa:x:1000:1000:Joachim Schroeder,,,:/home/joa:/bin/bash
joa:x:1000:1000:Joachim Schroeder,,,:/mynewhome:/bin/bash
```

Diese Variante ist zu bevorzugen, falls nicht alle Home-Verzeichnisse verschoben werden sollen. Für OpenWrt ist die Vorgehensweise grundsätzlich ähnlich, einige Eigenheiten sind jedoch zu beachten. So muss bspw. die Unterstützung für `ext3`-Dateiformate separat als Modul `kmod-fs-ext3` installiert werden. Weitere Details hierzu werden unter <http://wiki.openwrt.org/UsbStorageHowto> beschrieben.

Zu einer Anleitung für die Installation von OpenWrt-Paketen auf einem externen Speicherstick sei auf die Beschreibung unter <http://wiki.openwrt.org/PackagesOnExternalMediaHowTo> verwiesen.

<sup>37</sup> Diese Vorgehensweise wird auch unter <http://www.us.debian.org/doc/manuals/reference/ch-tips.en.html#s-archiving> empfohlen.

## Gerätetreiber und Kernelmodule

### 11.1 Einführung

In Linux-Systemen muss sich der Benutzer üblicherweise eher selten mit Details zur Hardware beschäftigen. Das Betriebssystem bildet eine abstrakte Schnittstelle, sodass bspw. das Abspeichern von Dateien auf Medien wie Festplatten, USB-Sticks oder Netzlaufwerken für den Anwender völlig identisch abläuft, obwohl die tatsächlichen Hardware-Zugriffe völlig unterschiedlich sind. Die Möglichkeit der Abstraktion ist nur mithilfe von Gerätetreibern möglich. Diese spielen im Betriebssystem eine wichtige Rolle, denn sie steuern eine Vielzahl unterschiedlicher Hardware-Komponenten an und verbergen dabei die komplexen Details. Für die meisten Geräte existieren solche Treiber, soll jedoch selbstentwickelte Hardware an ein Linux-System angeschlossen werden, muss sich der Anwender selbst um die Treiberentwicklung kümmern.

Ziel dieses Kapitels ist es, anhand einfacher Beispiele die Entwicklung von Gerätetreibern zu zeigen und darzulegen, wie Hardware auf verschiedenen Ebenen im Kernel angesprochen werden kann. Die Thematik ist relativ umfangreich, sodass dieses Kapitel nur einen geringen Teil der Treiberprogrammierung abdecken kann. Am Ende des Kapitels wird der Leser aber grundsätzlich in der Lage sein, mit den erworbenen Kenntnissen Treiber für selbstentwickelte Hardware zu programmieren.

Bestimmte Treiber, bspw. für den Zugriff auf die Festplatte, werden direkt nach dem Systemstart benötigt und müssen deshalb in den Kernel integriert werden. In der Summe stellen Gerätetreiber die umfangreichste Komponente im Kernel dar. Aus Gründen der Modularität und Performanz beinhaltet dieser jedoch nur die unbedingt notwendigen Gerätetreiber. Wird im Betrieb, z. B. durch das Anstecken einer Fotokamera an den USB-Anschluss, ein weiterer Gerätetreiber benötigt, so kann dieser zur Laufzeit nachgeladen werden. Der Treiber muss dazu in Form eines Kernelmoduls vorhanden sein. Wer

Gerätetreiber entwickeln möchte, der muss entsprechend zwangsläufig auch Kernelmodule programmieren.

Sollen Treiber für eigene Geräte entwickelt werden, so ist die Modulvariante einer festen Integration in den Kernel auf jeden Fall vorzuziehen. Einerseits müsste für einen Test sonst der gesamte Kernel jedesmal neu kompiliert werden. Andererseits ist es riskant, ein neu entwickeltes Modul mit in den Kernel aufzunehmen. Enthält das Modul noch schwerwiegende Fehler, so kann es vorkommen, dass der Rechner nicht mehr korrekt hochfährt oder instabil läuft – schlechte Voraussetzungen für die Fehlerbehebung.

Im diesem Kapitel wird ausschließlich Kernelversion 2.6 behandelt. Da der Kernel ständig weiterentwickelt wird und sich damit auch die Schnittstellen ändern, kann es sein, dass die mitgelieferten Beispiele zu einem späteren Zeitpunkt nicht mehr funktionsfähig sind. Die Änderungen sollten in den nächsten Jahren jedoch nicht gravierend sein. Die in diesem Kapitel angeführten Beispiele lassen sich unter Debian-Systemen und Puppy Linux direkt übersetzen und testen. Eine Verwendung in OpenWrt ist möglich, erfordert allerdings eine Integration in das Build-System wie sie Abschnitt 3.6 im Rahmen der IO-Warrior-Erweiterung beschreibt. Für das Einfügen und Entfernen von Kernelmodulen werden eine Reihe von Programmen benötigt, welche bei Debian im Paket `module-init-tools` enthalten sind.

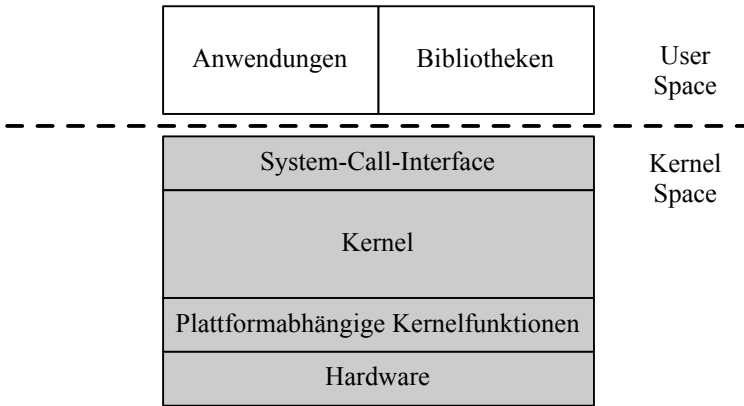
Abschnitt 11.2 beschäftigt sich ausführlicher mit der Systemarchitektur eines UNIX-Systems und einzelnen Komponenten des Kernels. Diese Grundlagen sind relativ theoretisch, erleichtern aber das Verständnis bei der Programmierung und helfen, Fehler zu vermeiden. Abschnitt 11.3 widmet sich dem Aufbau, Übersetzen und Debuggen von Kernelmodulen, bevor Abschnitt 11.4 in die Thematik der zeichenorientierten Gerätetreiber einführt. Abschnitt 11.5 erklärt, wie Dateioperationen als Schnittstelle für herkömmliche Anwendungen implementiert werden. Die verschiedenen Varianten der Hardware-Ansteuerung werden in Abschnitt 11.6 vorgestellt.

Alle für dieses Kapitel relevanten Quelltextbeispiele sind im Unterverzeichnis `<embedded-linux-dir>/examples/kernel-modules/` zu finden.

## 11.2 Grundlagen

### 11.2.1 Systemarchitektur

Bei der Programmierung von Gerätetreibern wird ein tiefer Eingriff in das System vorgenommen, welcher schwerwiegende Folgen haben kann. Um einen reibungsfreien Ablauf zu gewährleisten, sind Kenntnisse über den internen Aufbau des Systems wichtig. Diese sollten behandelt werden, bevor mit der eigentlichen Treiberprogrammierung begonnen wird.



**Abb. 11.1.** Komponenten eines UNIX-Betriebssystems.

Abbildung 11.1 zeigt vereinfacht die Architektur eines UNIX-Betriebssystems. Jedes UNIX-System besitzt einen *Kernel*, der als eigenständiges Programm bestimmte Dienste in Form von *System Calls* zur Verfügung stellt. Diese Dienste, wie bspw. das Ausgeben von Daten auf dem Bildschirm oder das Lesen und Schreiben von Dateien, können von anderen Anwendungen genutzt werden. Mit dem *System Call Interface (SCI)* bietet der Kernel eine einheitliche Schnittstelle, welche unabhängig von der verwendeten Hardware-Architektur ist und damit eine abstrahierende Schicht<sup>1</sup> bildet. Dies ist die Voraussetzung für die Software-Entwicklung auf Anwendungsebene für verschiedene Zielsysteme wie ARM, MIPS oder x86. Neben Anwendungen können auch Bibliotheksfunktionen, bspw. aus der GNU-C-Bibliothek, auf das SCI zugreifen. Um eine solche einheitliche Schnittstelle zu erhalten, müssen einige der Gerätetreiber für elementare Aufgaben wie Busansteuerung und die Verwendung von Prozessoren oder Timer-Bausteinen je nach Zielarchitektur entsprechend implementiert werden. Daraus folgt, dass ein Kernel immer für ein bestimmtes Zielsystem übersetzt wird und nur auf dieser Architektur lauffähig ist.

### *Kernel-Space und User-Space*

Eine Trennung der Speicherbereiche für Kernel und Anwendungen ist notwendig, um ein zuverlässig arbeitendes Betriebssystem zu erhalten. Nur so wird gewährleistet, dass Systemfunktionen unabhängig von den laufenden Anwendungen korrekt ausgeführt werden. Die Bezeichnungen *Kernel-Space* und *User-Space* beschreiben diese beiden Bereiche und werden oftmals für eine Unterscheidung der Ausführungsmodi verwendet. Während im Kernel-Space der uneingeschränkte Zugriff auf Hardware möglich ist, so geschieht dieser

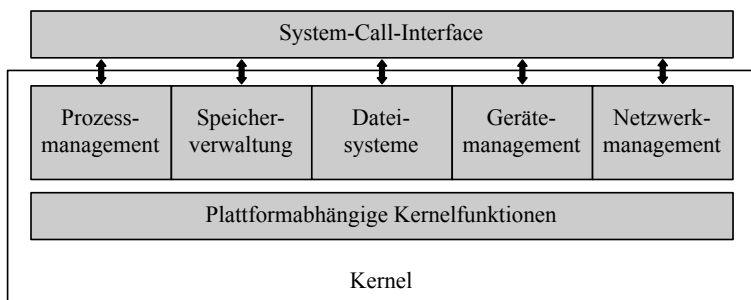
<sup>1</sup> Diese Schicht wird auch als *Hardware Abstraction Layer (HAL)* bezeichnet.

auf Anwendungsebene nur durch dedizierte System-Call-Funktionen. Dies bedeutet zwar gewisse Einschränkungen auf Anwendungsebene, garantiert aber auch ein zuverlässiges System. Nachgeladene Kernelmodule laufen ebenfalls im Kernel-Space, weshalb hier bei der Programmierung entsprechend viel Wert auf eine korrekte Nutzung der Ressourcen gelegt werden muss. Ein grundlegender Unterschied ist das Verhalten bei Fehlern in beiden Speichersegmenten. Während ein *Segmentation fault* im User-Space harmlos ist, kann ein *Kernel fault* den Prozess und das gesamte System lahmlegen, was eine Nachverfolgung mit einem Debugger unmöglich macht. Wie dennoch eine Fehlersuche im Kernel-Space erfolgen kann, wird in Abschnitt 11.3.3 erläutert.

### 11.2.2 Der Kernel

Abbildung 11.2 zeigt schematisch den Aufbau eines Kernels, welcher im Wesentlichen aus den folgenden Komponenten besteht:

- Prozessmanagement
- Speicherverwaltung
- Dateisysteme
- Gerätemanagement
- Netzwerkmanagement



**Abb. 11.2.** Vereinfachte Darstellung eines UNIX-Kernels.

Den einzelnen Komponenten kommen dabei folgende Aufgaben zu:

#### *Prozessmanagement*

Der Kernel startet, beendet, kontrolliert und versorgt Prozesse mit Daten (bspw. von der Tastatur) oder leitet diese weiter (bspw. an einen Bildschirm).



Auch die Kommunikation von Prozessen untereinander über Pipes oder Signale wird vom Prozessmanagement verwaltet. Es sorgt weiterhin dafür, dass auch auf einem System mit nur einem Prozessor Anwendungen quasi-parallel ausgeführt werden können. Hierbei wird jeder Anwendung eine bestimmte Rechenzeit zugeteilt und nach deren Ablauf zu einer anderen Anwendung gewechselt. Dies erfordert einen geringen Verwaltungsaufwand. Es ist jedoch insofern sinnvoll, da sonst Anwendungen, die auf Rückmeldungen des Systems oder einzelner Komponenten (Maus, Tastatur, Festplatte) warten, während dieser Dauer wertvolle Rechenzeit blockieren würden. Wird dieser Prozess<sup>2</sup> für eine absehbare Zeit schlafen gelegt und stattdessen ein anderer Prozess aufgeweckt, so kann die Rechenleistung wesentlich besser genutzt werden. Um die effiziente Festlegung der Wartezeiten und eine optimale Abarbeitungsreihenfolge kümmert sich der sog. *Scheduler*, der die einzelnen Zeitscheiben verteilt und damit Teil des Prozessmanagements ist. Der Vorgang selbst wird als *Scheduling* bezeichnet und in Kapitel 12 im Zusammenhang mit der Echtzeiterweiterung erörtert. Vorab nur soviel: Möchte man Einfluss auf die Zuteilung der Rechenzeit für kritische Aufgaben nehmen, so muss man beim Scheduler ansetzen. Bei einem Mehrprozessorsystem werden die Aufgaben vom Scheduler zusätzlich auf die einzelnen Prozessoren verteilt.

### *Speicherverwaltung*

Der Kernel stellt allen Prozessen Speicher in Form eines virtuellen Adressraums zur Verfügung. Dieser Adressraum besitzt eine logische Adressierung, d. h. die tatsächlichen physikalischen Adressen unterscheiden sich hiervon und werden nicht direkt angesprochen. Jede Anwendung bekommt Speicher in einem eigenen Segment zugeordnet, welches gegenüber dem Zugriff durch andere Anwendungen geschützt ist. Der Speicher des Kernels ist für Anwendungen tabu und deshalb ebenfalls in einem eigenen Segment untergebracht. Damit wird verhindert, dass Anwendungen aus dem User-Space fälschlicherweise auf den Kernel-Space zugreifen und dadurch Instabilitäten oder gar einen Systemabsturz verursachen. Der Kernel selbst kann ebenfalls nicht direkt auf Speicherbereiche einer Anwendung zugreifen. Ist dies erforderlich, so muss auf Funktionen zurückgegriffen werden, um den Zusammenhang zwischen logischen und physikalischen Adressen aufzulösen (vgl. Abschnitt 11.5.2).

### *Dateisysteme*

Unter UNIX kann fast jedes Medium als Datei behandelt werden, auch wenn es sich tatsächlich um RAM-Speicher, Festplattenspeicher oder um ein DVD-

<sup>2</sup> Ein Prozess kann entweder ein *Task* oder ein *Thread* sein. Ein *Task* ist ein Rechenprozess, der neben einem Codesegment und einem Stacksegment auch ein eigenes Datensegment besitzt. Teilt er sich dieses mit einem anderen Prozess, so bezeichnet man beide als *Threads*.

Laufwerk handelt. Der Kernel stellt trotz unterschiedlicher Hardware für alle Anwendungen ein einheitliches Erscheinungsbild des Systems in Form von Gerätedateien zur Verfügung. Diese sind üblicherweise aber nicht zwingend im System unter `/dev/` abgelegt.

Linux unterstützt relativ viele unterschiedliche Dateisysteme. Hierzu zählen unter anderem `vfat` (für USB-Sticks), `swap` (virtueller Arbeitsspeicher) oder `ext2/3` (Standard-Dateisystem unter Linux ohne/mit Journaling).

### *Gerätemanagement*

Wie bereits angesprochen werden Geräte unter UNIX als virtuelle Dateien behandelt. Soll nun auf eine solche Datei geschrieben werden, so müssen die Daten über diese Datei an das angeschlossene Gerät weitergeleitet werden. Diese Aufgabe übernehmen Gerätetreiber, die sich hinter den virtuellen Dateien verbergen und mit Kenntnis der genauen Funktionsweise ein reales Gerät ansteuern. Gerätetreiber sind dauerhaft oder temporär in den Kernel integriert und besitzen eine externe Schnittstelle, die für alle Module fest vorgegeben ist. Bei der internen Schnittstelle, welche die Arbeitsweise eines Gerätetreibers beschreibt, handelt es sich üblicherweise um einen von zwei verschiedenen Typen für den Zugriff auf zeichenorientierte Geräte (*Character-Devices*) oder blockorientierte Geräte (*Block-Devices*). Im ersten Fall werden die Daten zeichenweise verarbeitet. Dies geschieht unmittelbar, ist aber auch mit mehr Overhead verbunden. Im zweiten Fall werden die Daten zunächst gesammelt und in Blöcken verarbeitet, um größere Datenmengen effizienter zu handhaben. Ob ein Gerät zeichen- oder blockorientiert arbeitet, ist durch die Eingabe von `ls -l /dev/` ersichtlich: Bei zeichenorientierten Geräten steht an erster Stelle der Dateiattribut ein „c“, bei blockorientierten Geräten ein „b“. Für den Zugriff auf Multimediageräte reichen zeichen- und blockorientierte Schnittstellen nicht mehr aus, sodass dafür weitere Subsysteme wie SCSI- USB- oder I<sup>2</sup>C-Gerätetypen definiert wurden.

### *Netzwerkmanagement*

Die Netzwerkkommunikation betrifft üblicherweise mehrere Prozesse, findet asynchron statt und muss entsprechend zentral organisiert und verwaltet werden. Dies ist eine weitere Aufgabe des Kernels: Die Netzwerkpakete werden entgegengenommen, identifiziert, dispatcht und an Prozesse weitergeleitet. Alle dafür notwendigen Funktionen wie bspw. das Routing und die Adressauflösung sind ebenfalls im Kernel implementiert.

## 11.3 Programmierung von Kernelmodulen

Bevor mit der Programmierung begonnen werden kann, muss sichergestellt sein, dass die Kernelquellen für die passende Linuxversion vorhanden sind. Aufgrund der möglicherweise unterschiedlichen Datenstrukturen oder Funktionen zweier Kernelversionen ist es wichtig, dass Kernel und Modul die gleiche Versionsnummer besitzen. Module, welche mit Kernelquellen einer anderen Version übersetzt wurden, können nicht ohne Weiteres in einem System mit anderem Kernel geladen werden. Die Kernelversion kann mittels `uname -r` festgestellt werden. Die zugehörigen Kernelquellen werden unter Debian mit folgendem Befehl installiert:<sup>3</sup>

```
$ sudo apt-get install kernel-source
```

Die Kernelquellen werden auf dem System üblicherweise im Verzeichnis `/usr/src/linux-source-<version>` abgelegt. In vielen Makefiles wird auf `/lib/modules/<version>/build` referenziert, wobei dieser Ort wieder auf das Quellverzeichnis zeigt. Da unter Linux 2.6 die entwickelten Module gegen Objektdateien verlinkt werden, reichen die Kernel-Headers allein nicht mehr aus – hierin besteht ein wichtiger Unterschied zu früheren Linux-Versionen. Viele Compiler-Optionen werden durch die Konfiguration des Kernels (hinterlegt in `/usr/src/linux-source-<version>/config`) festgelegt. Sollte diese Konfigurationsdatei nicht vorhanden sein, so kann sie durch den Aufruf von `make menuconfig` im Quellverzeichnis initial erstellt werden.

### 11.3.1 Aufbau von Kernelmodulen

Kernelmodule bestehen immer aus mindestens zwei Funktionen: `init_module()` und `cleanup_module()`. Während die erste Funktion beim Laden des Moduls in den Kernel ausgeführt wird, macht die zweite beim Herausnehmen des Moduls aus dem Kernel in der Regel die Aktionen der Init-Routine rückgängig. Bei Linux 2.6 ist es möglich, die genannten Funktionen mit eigenen Namen zu versehen und eine Zuordnung über die Makros `module_init()` und `module_exit()` herzustellen. Das folgende Listing zeigt ein einfaches Hello-World-Modul mit diesen beiden Funktionen (vgl. Beispiel `helloworld` in `examples/kernel-modules/`):

```
#include <linux/version.h>  /* Linux Version */
#include <linux/module.h>    /* Makros and Defines */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Joachim Schroeder");
```

<sup>3</sup> Im OpenWrt-Buildsystem sind die Kernelquellen bereits enthalten. Kernelmodule müssen in das Buildsystem gemäß der Anleitung in Abschnitt 3.6 integriert werden. In Puppy-Linux werden die Kernelquellen durch Hinzufügen der Datei `kernel-src-<puppy-version>.sfs` installiert (vgl. Abschnitt 6.4).

```
MODULE_DESCRIPTION("Init-Exit Example");

int hello_init(void) {
    printk(KERN_INFO "init: Hello world!\n");

    return 0;
}

void hello_exit(void) {
    printk(KERN_INFO "exit: Bye, bye...\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Kernelmodule werden, wie auch der Kernel selbst, in der Programmiersprache C implementiert – eine Unterstützung von objektorientierten Ansätzen durch die Verwendung von C++ ist nicht möglich. Das zugrunde liegende objektbasierte Paradigma der Trennung von Code und Daten kann und sollte aber auch bei der Verwendung von C in Kernelmodulen berücksichtigt werden.

Über die include-Direktive `#include <linux/version.h>` wird die Kernelversion eingebunden, für die der Treiber kompiliert wird. Die zweite Header-Datei `<linux/module.h>` enthält Makros und Defines, welche für die Integration des Moduls in den Kernel notwendig sind. Hierin ist z. B. das Makro `MODULE_LICENSE` enthalten.

Über `MODULE_LICENSE` wird die Lizenz angegeben, unter welcher das Modul stehen soll. Dies ist ein wichtiger Schritt, um die Nutzung und Weiterverbreitung klarzustellen und um zukünftige Warnungen beim Laden eines Moduls zu verhindern. In diesem Fall wird, wie für alle Quelltexte dieses Buches, die *GNU General Public License* verwendet.<sup>4</sup> Die Makros `MODULE_AUTHOR` und `MODULE_DESCRIPTION` fügen weitere, optionale Angaben hinzu.

Die Funktion `printk()` ist im Kernel analog zur Funktion `printf()` im User-Space definiert. Für die Modulentwicklung steht keine Standard-C-Bibliothek zur Verfügung, da sich diese im User-Space befindet (vgl. Abbildung 11.1). Dieser Umstand soll hier nochmals verdeutlicht werden: Bei der Programmierung von Kernelmodulen wird nur gegen den Kernel selbst gelinkt. Die Verwendung der üblichen Bibliotheksfunktionen bzw. das Einbinden von Header-Dateien aus dem User-Space ist nicht möglich.

Über den String `KERN_INFO` wird die Priorität der Nachricht angegeben (Vorsicht: nach dem String wird kein Komma gesetzt). Die Ausgaben von `printk()` erfolgen auf die Kernel-Logs und können mit dem Befehl `dmesg` nachvollzogen werden.

---

<sup>4</sup> Die GPL-Lizenz besagt, dass jedes Programm, welches ganz oder nur in Teilen von anderen GPL-lizenzierten Quelltexten abgeleitet wird, ebenfalls unter die GPL zu stellen ist.

### 11.3.2 Übersetzung von Kernelmodulen

Um ein Kernelmodul zu übersetzen, muss der Compiler mit speziellen Optionen aufgerufen werden. Am einfachsten erfolgt die Übersetzung mithilfe eines Makefiles, welches für das Helloworld-Beispiel folgendermaßen aussieht:

```
obj-m += helloworld.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Die erste Zeile gibt an, dass es sich um ein Modul zum Nachladen in den Kernel handelt und nicht um einen fest in den Kernel integrierten Treiber (`obj-y += ...`). Im abgedruckten Makefile wird davon ausgegangen, dass der aktuelle Kernel (`uname -r`) der Kernel ist, für den das Modul erzeugt werden soll. Ist dies nicht der Fall oder liegen die Quellen nicht unter `/lib/modules/$(shell uname -r)/build`, so ist die Zeile entsprechend anzupassen. Ein Aufruf von `make` im Beispielvezeichnis übersetzt nun den Quelltext zu `helloworld.o` und linkt die Versionsinformationen aus `init/vermagic.o` hinzu. Es entsteht das fertige Modul `helloworld.ko`. Das Ziel `clean` ist übrigens optional und muss nicht angegeben werden. Der Befehl `modinfo` zeigt Versionsnummern und Abhängigkeiten des Moduls an:

```
$ modinfo helloworld.ko
filename:       helloworld.ko
description:    Init-Exit Example
author:         Joachim Schroeder
license:        GPL
srcversion:     BFD209FD94FF76D891A6617
depends:
vermagic:       2.6.24-19-generic SMP mod_unload 586
```

### 11.3.3 Test und Debugging

Nachdem das Kernelmodul übersetzt wurde, kann es über folgenden Befehl in den Kernel geladen werden:

```
$ sudo insmod helloworld.ko
```

Selbstverständlich sind für diese Aktion Root-Rechte erforderlich. Alternativ kann auch der Befehl `modprobe` verwendet werden, um ein Modul zu laden. Im Unterschied zu `insmod` werden bei `modprobe` etwaige Abhängigkeiten beachtet und u. U. auch andere Module zuvor geladen, um die benötigten Symbole bereitzustellen. Auch wenn das Beispiel noch keine Abhängigkeiten aufweist, so ist dies die sicherere Methode. Das fertige Modul `helloworld.ko` muss vor einem `modprobe`-Aufruf jedoch zunächst in das Modulverzeichnis kopiert werden. Dann wird mittels `depmod` eine Liste der Abhängigkeiten erstellt:

```
$ cp helloworld.ko /lib/modules/$(uname -r)
$ sudo depmod -a
$ sudo modprobe helloworld
```

Sollen bestimmte Kernelmodule bereits beim Start des Rechners automatisch geladen werden, so können die Modulnamen in der Datei `/etc/modules` hinzugefügt werden. Um zu überprüfen, ob das Modul tatsächlich eingefügt wurde, liefert `lsmod` eine Liste aller momentan geladenen Kernelmodule:

```
$ lsmod
Module                Size  Used by
helloworld             2560    0
af_packet             23812    2
rfcomm                 41744    2
l2cap                  25728   13 rfcomm
...
```

Bislang wurden nur die `init`- und `exit`-Funktionen implementiert. Als einzige weitere Aktion bleibt daher das Entfernen des Moduls aus dem Kernel:

```
$ sudo rmmod helloworld
```

Es ist anzumerken, dass ein Entfernen von Modulen nur dann möglich ist, wenn keine Anwendung aus dem User-Space den Treiber aktuell verwendet. Dies wäre der Fall, wenn bspw. eine zugehörige Gerätedatei noch geöffnet ist. Die Voraussetzung hierfür ist, dass das Modul und die Anwendung korrekt implementiert wurden. Programmierfehler (bspw. ein vergessener Aufruf von `close()` in der Anwendung) können dazu führen, dass das Modul nicht mehr entladen werden kann und sich nur durch einen System-Neustart aus dem Kernel entfernen lässt. Die Ausgaben des Moduls werden in der Datei `/var/log/messages` abgelegt, können aber auch über einen Aufruf von `dmesg` nachvollzogen werden. Die folgende Ausgabe zeigt die letzten fünf Zeilen eines `dmesg`-Aufrufs:

```
$ dmesg | tail -n5
[ 66.909291] Bluetooth: RFCOMM ver 1.8
[ 70.902564] NET: Registered protocol family 17
[ 83.043568] eth0: no IPv6 routers present
[230584.548791] init: Hello world!
[230607.917271] exit: Bye, bye...
```

Es ist erkennbar, dass das Modul korrekt geladen und wieder entfernt wurde. Ein fertiger Kernaltreiber kann in einer eigenen User-Space-Anwendung verwendet und hierüber auch intensiv getestet werden. Um die grundlegende Funktionalität zu erproben, lassen sich vorhandene Systemprogramme wie bspw. `echo`, `cat` oder `less` nutzen. Wurden für einen Treiber Dateioperationen implementiert und eine Gerätedatei `/dev/helloworld/` erzeugt (vgl. Abschnitt 11.5), so ruft folgender Befehl die Funktionen `open()`, `read()` und `release()` des Moduls auf:

```
$ cat /dev/helloworld
```

Sollen Daten an das Kernelmodul gesendet werden, so kann dies über folgenden Befehl geschehen:

```
$ echo "Hallo" > /dev/helloworld
```

Die Fehlersuche gestaltet sich in Kernelmodulen wesentlich komplexer als in Anwendungen, da Kernel-Debugging standardmäßig nicht unterstützt wird. Der *Kerneldebugger*<sup>5</sup> ist momentan noch nicht im Kernel enthalten, kann aber über Kernel-Patches nachgerüstet werden. Ein Nachteil bei der Verwendung des Kerneldebuggers ist die Notwendigkeit eines zweiten Rechners. Auf dem Zielrechner läuft der zu debuggende Linux-Kernel, auf dem Hostrechner der Debugger selbst. Beide Rechner werden über eine serielle Schnittstelle verbunden. Wenn der Debugger die Netzwerktreiber für das Zielsystem enthält, so kann in Einzelfällen auch Ethernet verwendet werden. Ein weiterer Nachteil dieses Verfahrens ist, dass der Kerneldebugger nicht für alle Hardware-Plattformen zur Verfügung steht.

Die wohl einfachste Methode um Debugging-Funktionalität zu erhalten, ist die Ausgabe von Kommentaren mittels `printk()` wie sie im ersten Beispiel gezeigt wurde. Der Prozess `syslogd` protokolliert wichtige Ereignisse des Systems entsprechend der in `/etc/syslog.conf` hinterlegten Konfiguration. Dort kann spezifiziert werden, ob Ereignisse nach Bereichen wie *Mailsystem*, *Kernel*, oder *Prioritäten* aufgeteilt werden. Weiterhin kann ein Speicherort für diese Nachrichten vorgegeben werden. Die folgende Zeile legt fest, dass sämtliche Ereignisse in der Datei `/var/log/messages` protokolliert werden:

```
*. * -/var/log/messages
```

Bei der Verwendung von `printk()` ist es sinnvoll, die Klassifizierung der Meldungen zu nutzen (vgl. Tabelle 11.1) und für Debug-Nachrichten die Priorität `KERN_DEBUG` zu verwenden. Sollen für eine Release-Version Debug-Meldungen abgeschaltet werden, so können die Ausgaben in Präprozessor-Anweisungen der folgenden Art gekapselt werden:

```
#ifdef DEBUG
    printk(KERN_DEBUG "Dies ist eine Debug-Nachricht\n");
#endif
```

Eine Berücksichtigung erfolgt dann nur mit Angabe der Compileroption `-DDEBUG`. Compiler-Flags werden in der Variablen `EXTRA_CFLAGS` gesetzt und im Makefile angegeben:

```
EXTRA_CFLAGS += -DDEBUG
obj-m += helloworld.o
...
```

Aus Gründen der Übersichtlichkeit kann auf die Makros `pr_debug()` und `pr_info()` zurückgegriffen werden, welche in `linux/kernel.h` enthalten sind

<sup>5</sup> Vergleiche auch die URL <http://kgdb.sourceforge.net/>. Eine Dokumentation zum Kerneldebugger befindet sich im Quellcode-Verzeichnis des Kernels unter `Documentation/i386/kgdb/`.

und bei gesetztem Debug-Flag eine `printk()`-Anweisung mit korrekter Priorität erzeugen.

```
#include <linux/kernel.h>
pr_debug("Dies ist eine Debug-Nachricht\n");
```

Bezeichnung	Wert	Bedeutung
KERN_EMERG	< 0 >	Das System ist nicht mehr nutzbar
KERN_ALERT	< 1 >	Der aktuelle Zustand erfordert sofortige Maßnahmen
KERN_CRIT	< 2 >	Der Systemzustand ist kritisch
KERN_ERR	< 3 >	Fehlerzustände sind aufgetreten
KERN_WARNING	< 4 >	Warnung
KERN_NOTICE	< 5 >	Wichtige Nachricht, aber kein Fehlerzustand
KERN_INFO	< 6 >	Information
KERN_DEBUG	< 7 >	Debug-Informationen

Tabelle 11.1. Prioritäten der Kernelnachrichten.

11.3.4 Übergabe von Kommandozeilenparametern

Ähnlich wie Anwendungen können auch Kernelmodulen beim Laden Argumente übergeben werden. Im Gegensatz dazu geschieht dies jedoch nicht über `argc` und `argv`, sondern über eine vorab fest definierte Schnittstelle. Statisch deklarierte Variablen können mit dem Makro `module_param()` (definiert in `linux/moduleparam.h`) als Übergabeargument gekennzeichnet und dem Kernel bekannt gemacht werden. Neben dem Variablennamen werden `module_param()` der Variablentyp und die Zugriffsrechte der Parameterdatei im `sys`-Dateisystem mitgegeben. Über das Makro `MODULE_PARM_DESC` lässt sich eine Bezeichnung für den jeweiligen Parameter setzen, welche in der Modulbeschreibung hinterlegt wird (vgl. Befehl `modinfo`):

```
static int arg1 = -1;
static char* arg2 = "isempty";
static short arg3[2] = {0,0};

module_param(arg1, int,0);
MODULE_PARM_DESC(arg1, "is a int");
module_param(arg2, charp,0);
MODULE_PARM_DESC(arg2, "is a string");
module_param_array(arg3, short, &count, 0);
MODULE_PARM_DESC(arg3, "is a short-array");
```

Die Angabe eines Arrays als Kommandozeilenargument ist ebenfalls möglich. Eine Übergabe geschieht mit dem Makro `module_param_array()`, welches als drittes Argument die Anzahl der übermittelten Array-Einträge erhält. Diese können in einer separaten Variable aufgefangen werden. Bei der Angabe zu



vieler Argumente lässt sich das Modul nicht laden und es folgt die Fehlermeldung `-1 Invalid parameters`. Das Beispiel `commandline` zeigt die Verwendung von Kommandozeilenargumenten. Die korrekte Übernahme der bei einem Aufruf spezifizierten Argumente lässt sich wiederum mit dem Befehl `dmesg` verifizieren:

```
$ sudo insmod commandline.ko arg1="42" arg2="nice_string" arg3="2,4"
$ dmesg
...
[1579673.957959] arg1 is of type int: 42
[1579673.957962] arg2 is of type char*: nice_string
[1579673.957964] arg3 is of type short array and contains: 2 elements
[1579673.957967] arg3[0] = 2
[1579673.957969] arg3[1] = 4
```

## 11.4 Zeichenorientierte Gerätetreiber

In diesem Abschnitt wird ein kompletter, zeichenorientierter Gerätetreiber entwickelt, über den mit IO-Funktionen im User-Space kommuniziert werden kann. Die Entwicklung von Treibern für Block- oder Netzwerkgeräte ist im Rahmen von Embedded-Anwendungen eher weniger relevant und wird deshalb an dieser Stelle nicht behandelt. Interessierte Leser finden in [Quade 06] eine gute Einführung in die Thematik der Linux-Treiberentwicklung, die sich auch den speziellen Geräteklassen widmet.

### 11.4.1 Major-, Minor- und Geräteummern

Wie bereits erwähnt, werden Geräte unter UNIX über das Dateisystem angesprochen. Die dafür verwendeten Gerätedateien befinden sich üblicherweise im Verzeichnis `/dev/` und beschreiben den Gerätetyp durch den ersten Buchstaben in der Ausgabe von `ls -l /dev/`:

```
$ ls -l /dev/
...
brw-rw---- 1 root disk      1,    0 Jun 23 14:01 ram0
...
crw----- 1 root root      4,   64 Jun 23 14:04 ttyS0
crw-rw---- 1 root dialout   4,   65 Jun 23 14:01 ttyS1
...
```

Ein „c“ steht für ein zeichenorientiertes Gerät, ein „b“ für ein blockorientiertes Gerät. In der fünften und sechsten Spalte für Eigentümer und Gruppe folgen zwei Nummern: Die *Major- und Minornummer* eines Gerätes. Die Majornummer beschreibt den Treiber, der für den Zugriff auf das Gerätes verwendet wird. Registriert sich ein Treiber beim System, so wird ihm eine Majornummer zugeteilt und ein entsprechender Eintrag in `/proc/devices` erzeugt.

Die Minornummer identifiziert das Gerät selbst, welches in Kombination mit einem bestimmten Treiber arbeitet. Bei den beiden seriellen Schnittstellen `/dev/ttyS<0,1>` im obigen Beispiel handelt es sich um unterschiedliche Geräte, für die aber der gleiche Treiber verwendet wird. Dem Treiber wird die Minornummer bei einem Zugriff übermittelt und damit das angeschlossene Gerät bekannt gemacht. Nur so kann der Treiber das Gerät mit der korrekten physikalischen Adresse ansprechen.

Der Kernel arbeitet intern seit Version 2.6 nicht mehr mit Major- und Minornummern, sondern mit Gerätenummern vom Typ `dev_t` (definiert in `linux/types.h`). Von dieser 32-Bit-Variablen werden die oberen 12 Bit für die Majornummer, die unteren 20 Bit für die Minornummer reserviert. Damit können nun jeweils mehr als 255 Treiber und Geräte verwendet werden. Voraussetzung ist allerdings, dass der Treiber ebenfalls auf Gerätenummern und nicht mehr auf Major- und Minornummern basiert. Aus Kompatibilitätsgründen zu späteren Versionen und für eine effiziente Ressourcennutzung sollten Gerätenummern verwendet werden (auch wenn momentan nur maximal 255 Minornummern genutzt werden können). Konkret heißt dies, dass in diesem Beispiel zur Registrierung der Geräte die folgenden, neueren Funktionen verwendet werden:

```
int register_chrdev_region(dev_t from, unsigned count, char
    *name);
```

Diese Funktion reserviert eine aufeinanderfolgende Anzahl von Gerätenummern für zeichenorientierte Geräte, die das Kernelmodul selbst vorgibt. `from` enthält dabei die erste Nummer der gewünschten Folge (mit gültiger Majornummer), `count` die Anzahl an Gerätenummern. `name` gibt an, unter welcher Bezeichnung die Nummernfolge im Kernel geführt wird. Bei erfolgreicher Ausführung wird 0 zurückgeliefert, ansonsten ein negativer Fehlercode.

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned
    count, char *name);
```

Diese Funktion reserviert `count` Gerätenummern in einem freien Bereich und legt die erste Nummer im Parameter `dev` ab. Die Reservierung wird im Kernel als `name` referenziert. Mit `baseminor` wird die erste zu verwendende Minornummer angegeben. Diese Funktion sollte zum Einsatz kommen, falls nicht sicher ist, welche Gerätenummern frei verfügbar sind.

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

Die Funktion bewirkt eine Freigabe von `count` Gerätenummern ab der Stelle `from`.

Wie eine Registrierung mithilfe dieser Funktionen durchgeführt wird, zeigt Abschnitt 11.4.2. Da auch gegenwärtig noch viele Treiber die ältere Registrierungsvariante enthalten, wird dieses Verfahren in Abschnitt 11.4.3 der Vollständigkeit halber erklärt. Für die sichere Umrechnung von

Gerätenummern in Major- und Minornummern stehen die folgenden Makros zur Verfügung:

- `MAJOR(dev_t dev)`: Berechnet die Majornummer zu einer Gerätenummer.
- `MINOR(dev_t dev)`: Berechnet die Minornummer zu einer Gerätenummer.
- `MKDEV(int major, int minor)`: Liefert die Gerätenummer zu `major` und `minor`.

### 11.4.2 Modul-Registrierung

In den bisherigen Beispielen wurden bereits Kernelmodule entwickelt, geladen und mit Attributen versehen. Um ein Modul als Treiber verwenden zu können, muss es beim IO-Management des Systems angemeldet werden. Die Funktionen `(un)register_chrdev_region()` wurden bereits im letzten Abschnitt besprochen und dienen dazu, Gerätenummern vom System anzufordern und wieder freizugeben – in diesem Fall mit einer fest vorgegebenen Majornummer. Zeichenorientierte Gerätetreiber werden im Kernel mit einer Struktur vom Typ `struct cdev` repräsentiert (deklariert in `linux/cdev.h`). Bevor ein Gerätetreiber verwendet werden kann, müssen dem Kernel Informationen über den Treiber in Form dieser Struktur mitgeteilt werden (ein Beispiel hierfür sind mögliche Dateioperationen). Dafür stehen folgende Funktionen zur Verfügung:

```
struct cdev *cdev_alloc(void);
```

Diese Funktion instanziiert eine Struktur vom Typ `struct cdev` und gibt im Erfolgsfall die Adresse zurück, sonst `NULL`.

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Die Funktion führt die Initialisierung der Struktur `cdev` durch und ordnet dieser die in `fops` definierten Dateioperationen zu (vgl. Abschnitt 11.5).

```
int cdev_add(struct cdev *drv, dev_t num, unsigned count);
```

Die Funktion registriert einen zeichenorientierten Treiber beim Kernel. Die Informationen dazu müssen zuvor in `drv` hinterlegt werden. `num` gibt die erste Gerätenummer an, `count` die Anzahl der Geräte, die der Treiber verwalten soll. Bei Erfolg wird 0 zurückgeliefert, sonst ein Fehlercode.

```
void cdev_del(struct cdev *drv);
```

Die Funktion entfernt die Struktur `drv` und meldet damit den zeichenorientierten Treiber wieder beim Kernel ab.

Der Ablauf der Prozeduren zur An- und Abmeldung von Gerätetreibern inklusive Registrierung der Gerätenummern wird im Beispiel `module_reg_new` veranschaulicht. Hierbei handelt es sich um eine Treiberregistrierung auf Basis von Gerätenummern, nicht nur auf Basis einer Majornummer (vgl. Abschnitt 11.4.3). Unter Verwendung der vorgestellten Funktionen könnten Routinen zum An- und Abmelden des Gerätetreibers folgendermaßen aussehen:

```

#include <linux/version.h>  /* Linux Version */
#include <linux/module.h>   /* Makros and Defines */
#include <linux/fs.h>
#include <linux/cdev.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Joachim Schroeder");
MODULE_DESCRIPTION("Module Registration Argument Example");

#define MAJORNUM 60
#define NUMDEVICES 1
#define DEVNAME "myfirstdevice"

static struct cdev *driver_info = NULL;
static struct file_operations fops;

int modulereg_init(void) {
    printk(KERN_INFO "Module modulereg: init()\n");

    // allocate device number
    if( register_chrdev_region( MKDEV(MAJORNUM,0), NUMDEVICES, DEVNAME ) ) {
        pr_debug("Device number 0x%x not available ...\n", MKDEV(MAJORNUM,0));
        return -EIO;
    }

    // allocate device
    driver_info = cdev_alloc();
    if( driver_info == NULL ) {
        pr_debug("cdev_alloc failed!\n");
        goto free_devnum;
    }

    // specify device structure (init) and register (add)
    kobject_set_name(&driver_info->kobj, DEVNAME );
    driver_info->owner = THIS_MODULE;
    cdev_init( driver_info, &fops );
    if( cdev_add( driver_info, MKDEV(MAJORNUM,0), NUMDEVICES) ) {
        pr_debug("cdev_add failed!\n");
        goto free_cdev;
    }
    return 0;

free_cdev:
    kobject_put(&driver_info->kobj);
    driver_info = NULL;
free_devnum:
    unregister_chrdev_region(MKDEV(MAJORNUM,0), NUMDEVICES);
    return -1;
}

void modulereg_exit(void) {
    printk(KERN_INFO "Module modulereg: exit()\n");

    // remove char device from system
    if( driver_info ) cdev_del( driver_info );

    // free device number
    unregister_chrdev_region(MKDEV(MAJORNUM,0), NUMDEVICES);
}

module_init(modulereg_init);
module_exit(modulereg_exit);

```

In der Initialisierungsroutine `modulereg_init()` wird dem Treiber zunächst eine Geräteummer zugeteilt, die sich aus der Majornummer `majornum` berech-

net – hier soll eine einzelne Nummer für den Anfang genügen. Ist dies erfolgreich, so wird im nächsten Schritt über `cdev_alloc()` eine Struktur mit den Treiberinformationen angelegt. Vor der Initialisierung mittels `cdev_init()`, bei der unter anderem die in `fops` hinterlegten Dateioperationen zugewiesen werden, kann die Variable `owner` auf das Symbol `THIS_MODULE` gesetzt werden. Durch diese Zuweisung wird ein Entladen des Moduls verhindert, solange eine Anwendung den Treiber verwendet. Nachdem die Treiberstruktur mittels `cdev_init()` initialisiert wurde, kann der Treiber nun mit der Funktion `cdev_add()` beim Kernel angemeldet werden.

Falls eine dieser Funktionen fehlschlägt und damit `modulereg_init()` den Wert `-1` zurückliefert, wird das Modul nicht geladen. In diesem Fall können in der `exit`-Routine keine Funktionsaufrufe zum Entfernen des Treibers aus dem Kernel und zur Freigabe der Gerätenummer erfolgen. Diese Aufrufe müssen deshalb unmittelbar nach dem Auftreten des Fehlers ausgeführt werden und sind am einfachsten über `goto`-Sprunganweisungen zu realisieren. Wenn auch sonst aus gutem Grund verpönt, so ist die `goto`-Verwendung in diesem Fall ausnahmsweise berechtigt.

Wohlgermerkt sind bis jetzt noch keine Dateioperationen implementiert, da zunächst nur die Registrierung behandelt werden soll. Der Bereitstellung von Dateioperationen widmet sich Abschnitt 11.5. Wird das Beispielm Modul `module_reg_new` übersetzt und geladen, so taucht der Treiber samt Majornummer als zeichenorientiertes Gerät in `/proc/devices` auf:

```
$ cat /proc/devices
Character devices:
...
29 fb
60 myfirstdevice
99 ppdev
116 alsa
...
```

Um den Treiber verwenden zu können wird eine Gerätedatei benötigt. Mit dem Programm `mknod` können spezielle Block- oder Zeichengeräte erzeugt werden. Der folgende Befehl erstellt eine solche Spezialdatei für ein zeichenorientiertes Gerät mit Majornummer 60 und Minornummer 0:

```
$ sudo mknod /dev/myfirstdevice -m 666 c 60 0
```

Soll die Spezialdatei im Verzeichnis `/dev/` angelegt werden, so sind Root-Rechte notwendig. Die Option `-m 666` setzt den Zugriffsmodus und vergibt in diesem Fall Lese- und Schreibrechte für alle Teilnehmer. Wird dafür kein Wert angegeben, so werden die Zugriffsrechte auf die Datei aus der Bitdifferenz von 0666 und der aktuellen `umask` des aufrufenden Prozesses gebildet. Im Anhang geben die Abschnitte A.4.3 und A.4.4 weiterführende Informationen zu `mknod` und zur Vergabe von Zugriffsrechten.

Nun ist zwar eine Gerätedatei vorhanden, diese bringt aber keinen Nutzen ohne IO-Funktionen. Abschnitt 11.5 zeigt, wie dem Treiber diese Funktionen hinzugefügt und in einer Anwendung verwendet werden.

### 11.4.3 Gerätetreiber-Registrierung nach alter Schule

In vielen aktuellen Treibern wird noch immer die *alte* Möglichkeit der Treiberregistrierung verwendet. Auch wenn diese Funktionen längerfristig wohl aus dem Kernel verschwinden werden, so ist dies Grund genug, die Vorgehensweise kurz zu erläutern: Um einen Treiber nach dem alten Schema zu registrieren, wird innerhalb der Initialisierungsroutine die Funktion `register_chrdev()` aufgerufen (vgl. Beispiel `module_reg_old`). Die Funktion meldet das Modul als zeichenorientierten Gerätetreiber mit Majornummer `major` unter dem Namen `name` an. Eine Verwendung der neueren Gerätenummern ist nicht möglich.

```
#include <linux/fs.h>
int register_chrdev(unsigned int major, const char *name, struct
    file_operations *fops);
```

Wird als Majornummer 0 übergeben, so vergibt das IO-Subsystem eine dynamische Nummer. In der Datei `/proc/devices` taucht der Treiber unter angegebener Bezeichnung `name` mit zugehöriger Majornummer auf. Diese muss dort nachgelesen werden, falls eine dynamische Zuteilung verwendet wird. Mit dem Zeiger `fops` auf eine Struktur vom Typ `file_operations` (deklariert in `linux/fs.h`) wird die Treiberschnittstelle festgelegt (vgl. auch Abschnitt 11.5). Ein negativer Rückgabewert bedeutet, dass der Treiber vom System nicht akzeptiert wurde. Dies geschieht dann, wenn die angegebene Majornummer bereits vergeben wurde oder keine neue Majornummer verfügbar ist.

Es ist zu beachten, dass die Majornummer für jeden Treiber eindeutig sein muss und viele Nummern bereits fest vergeben sind.<sup>6</sup> Für eigene Entwicklungen stehen die Bereiche 60–63, 120–127 und 240–254 zur Verfügung. Über die folgende Funktion muss jeder Treiber beim Beenden des Moduls unbedingt wieder vom System abgemeldet werden:

```
#include <linux/fs.h>
int unregister_chrdev(unsigned int major, const char *name);
```

Der Aufruf ist entsprechend in der `exit`-Routine zu platzieren. Der Treiber wird anhand der Majornummer `major` und seinem Namen `name` identifiziert und kann nur abgemeldet werden, sofern diese Angaben mit jenen in `register_chrdev()` übereinstimmen. Wird ein Treiber nicht korrekt aus dem System entfernt, so steht die Majornummer erst nach einem Neustart wieder zur Verfügung.

<sup>6</sup> Nachzulesen in der Datei `/usr/src/linux/Documentation/devices.txt`.

## 11.5 Implementierung von Dateioperationen

### 11.5.1 Die Struktur `file_operations`

In Abschnitt 11.2 wurde erklärt, dass Anwendungen im User-Space über ein sog. *System Call Interface* auf Funktionen im Kernel zugreifen können. Die Schnittstelle zu einem Treiber wird als Teil des gesamten SCI in Form einer Struktur vom Typ `file_operations` vorgegeben (definiert in `linux/fs.h`). Die in dieser Struktur enthaltenen Funktionszeiger repräsentieren Dateioperationen oder auch Einstiegspunkte in den Treiber, hinter denen sich vom Anwender implementierte System-Call-Funktionen verbergen. Da die Struktur relativ umfangreich ist, sollen hier nur die wesentlichen Einträge erklärt werden:

```
struct module *owner;
```

Bei diesem Feld handelt es sich nicht um einen Zeiger auf eine Dateioperation, sondern um den Zeiger auf eine Struktur vom Typ `module`, der die `fops`-Struktur gehört. Damit werden üblicherweise Nutzungsrechte hinterlegt, um das Entladen eines noch in Verwendung befindlichen Moduls zu verhindern. Wird das Modul in keinem anderen Modul verwendet, so sollte der Wert auf das in `linux/module.h` definierte Makro `THIS_MODULE` gesetzt werden.

```
ssize_t (*read) (struct file *instanz, char __user *userbuf,
                 size_t num, loff_t *off);
```

Fordert die Treiberinstanz `instanz` auf, `num` Daten-Bytes im User-Space an der Stelle `userbuf` abzulegen. Über `off` kann ein Offset angegeben werden. `instanz` beinhaltet Informationen über das logische Gerät und den Zugriffsmodus (blockierend, nicht-blockierend). Wurde dieser Funktionszeiger nicht gesetzt, so wird bei einem Aufruf der Wert `-EINVAL` (Invalid argument) zurückgegeben.

```
ssize_t (*write) (struct file *instanz, const char __user
                  *userbuf, size_t num, loff_t *off);
```

Fordert die Treiberinstanz `instanz` auf, `num` Daten-Bytes zu schreiben, welche an der Stelle `userbuf` im User-Space bereitgestellt sind. Über `off` kann ein Offset angegeben werden. Wurde dieser Funktionszeiger nicht gesetzt, so wird bei einem Aufruf `-EINVAL` (*Invalid argument*) zurückgegeben.

```
int (*ioctl) (struct inode *node, struct file *instanz,
              unsigned int cmd, unsigned long arg);
```

Führt eine gerätespezifische Anweisung aus, die nicht einheitlich formuliert werden kann. `node` enthält alle Angaben einer zugehörigen Gerätedatei wie bspw. Zugriffsrechte und Informationen über deren Besitzer. Ein

Zeiger auf die Treiberinstanz ist in `instanz` enthalten. Das IO-Control-Kommando wird in `cmd` übergeben, ein Argument in `arg`. Dabei kann es sich auch um einen Zeiger auf eine Struktur von Argumenten handeln. Ist diese Funktion nicht vorhanden, so wird vom System `-ENOTTY` (No such ioctl for device) zurückgeliefert.

```
int (*open) (struct inode *node, struct file *instanz);
```

Öffnet eine Gerätedatei `node` und meldet das Gerät beim Treiber `instanz` an. Ist die Aktion erfolgreich, so wird 0 zurückgegeben, sonst ein Fehlercode (bspw. bei unzureichenden Zugriffsrechten auf `node`). Ein Aufruf von `open()` erfolgt in der Anwendung vor allen anderen Dateioperationen.

```
int (*release) (struct inode *node, struct file *instanz);
```

Diese Funktion wird durch den Aufruf von `close()` anwendungsseitig ausgeführt und gibt üblicherweise die Ressourcen der Treiberinstanz `instanz` frei. Ist dies erfolgreich, so wird 0 zurückgegeben.

Aus Kompatibilitätsgründen sollte für das Füllen der Struktur immer die explizite Syntax gewählt werden bzw. eine Angabe des jeweiligen Variablennamens stattfinden. Aus den gleichen Gründen ist der GNU-Erweiterung (bspw. `read: fops_read`) die C99-Syntax<sup>7</sup> vorzuziehen:

```
static struct file_operations fops = {
    .owner      = THIS_MODULE,
    .read       = fops_read,
    .write      = fops_write,
    .ioctl      = fops_ioctl,
    .open       = fops_open,
    .release    = fops_close
};
```

Hierbei müssen nicht alle Felder belegt werden – nicht benötigte Zeiger werden automatisch mit 0 initialisiert. Üblicherweise wird für eine Instanz dieser Struktur der Name `fops` verwendet. Beim Aufruf eines System-Calls im User-Space wird dieser an die zugehörige Funktion im Treiber weitergeleitet. Die Felder in der Struktur `file_operations` repräsentieren dabei die gleichnamigen Funktionen im User-Space, eine Ausnahme stellt nur `.release` dar. Diese Variable wird referenziert, wenn die Anwendung ein Gerät mit dem Befehl `close()` schließt. Im Beispiel `file_operations` wird die Implementierung von Dateioperationen gezeigt. Zuvor wird der Datentransfer zwischen Kernel-Space und User-Space kurz diskutiert.

### 11.5.2 Kopieren von Daten zwischen Kernel- und User-Space

Die mit einem realen Gerät kommunizierten Daten gelangen zunächst in den Kernaltreiber und müssen bei einer Leseoperation in den User-Space transfe-

<sup>7</sup> Mit C99 wird die überarbeitete Spezifikation des ISO/IEC-Standards der Programmiersprache C bezeichnet.



riert bzw. bei einer Schreiboperation in den Kernel-Space übertragen werden. Um Daten zwischen diesen Speichersegmenten zu kopieren, stehen folgende Funktionen zur Verfügung (definiert in `asm/uaccess.h`):

```
int get_user(var, src_ptr);
```

Dieses Makro kopiert einen einzelnen Datenblock (1, 2, 4 oder 8 Bytes) von der Adresse `src_ptr` in die Variable `var` im Kernel-Space. Die Größe der zu übertragenden Daten hängt vom Typ des `ptr`-Argumentes ab. Im Erfolgsfall wird 0, ansonsten `-EFAULT` zurückgegeben.

```
int put_user(var, dest_ptr);
```

Kopiert einen Datenblock (1, 2, 4 oder 8 Bytes) aus der Variablen `var` an die Adresse `dest_ptr` im User-Space. Die Größe der zu übertragenden Daten hängt vom Typ des `ptr`-Arguments ab. Im Erfolgsfall wird 0, ansonsten `-EFAULT` zurückgegeben.

```
unsigned long copy_from_user(void *to, const void *from,
    unsigned long num);
```

Kopiert ab der Speicherstelle `from` aus dem User-Space `num` Daten-Bytes an Adresse `to` im Kernel-Space. Als Rückgabewert wird die Anzahl der nicht kopierten Daten-Bytes geliefert, im Erfolgsfall also 0.

```
unsigned long copy_to_user(void *to, const void *from, unsigned
    long num);
```

Kopiert ab der Speicherstelle `from` aus dem Kernel-Space `num` Daten-Bytes an Adresse `to` im User-Space. Als Rückgabewert wird die Anzahl der nicht kopierten Daten-Bytes geliefert, im Erfolgsfall also 0.

Den genannten Funktionen ist gemein, dass sie den angegebenen Speicherbereich auf Gültigkeit überprüfen. Dies wird durch einen impliziten Aufruf von `access_ok()` erreicht. In Einzelfällen kann es effizienter sein, diesen Test selbst durchzuführen, um anschließend mit den Funktionen `__get_user()`, `__put_user()` bzw. `__copy_from_user()`, `__copy_to_user()` Daten ohne Überprüfung zu kopieren. Der Einsatz dieser Funktionen ist jedoch mit Vorsicht zu genießen und bringt nur bei bestimmten Übertragungsvarianten Vorteile – z. B. wenn in einer `read`-Funktion mehrere Werte hintereinander mittels `put_user()` übertragen werden. Der einmalige Aufruf von `access_ok()` vor der ersten Übertragung mit `__put_user()` würde dann genügen. Die Makros `get_user()` und `put_user()` sind sehr schnell und sollten der Verwendung von `copy_to/from_user()` vorgezogen werden, falls nur einzelne Werte übertragen werden.

Im Beispiel `file_operations` wird die Implementierung von Treiberfunktionen und der Datentransfer zwischen Kernel- und User-Space anhand eines Zeichenpuffers gezeigt. Der Anwender kann über Lese- und Schreiboperationen auf den Puffer zugreifen und diesen über `ioctl`-Funktionen leeren oder mit einem Standardtext füllen. Die Initialisierungsroutinen für das Kernelmodul sind dem Beispiel aus `modulereg_new` entnommen und werden im folgenden

Listing deshalb nicht nochmals aufgeführt. Über das Makro `COPY_MULTI_BYTES` kann zwischen den zwei Varianten der Übertragung mittels `put/get_user()` bzw. `copy_to/from_user()` gewechselt werden. Der folgende Auszug zeigt die Implementierungen der Dateioperationen (die `ioctl()`-Funktion wird erst in Abschnitt 11.5.3 behandelt):

```
/* FOPS Functions */
static int fops_open(struct inode* inode, struct file* file){

    if (is_open) return -EBUSY;

    is_open++;
    p_msg = msg;
    try_module_get(THIS_MODULE);
    pr_debug("Module fops: device %s was opened from device with minor no %d\n", DEVNAME, iminor(inode));
    return 0;
}

static int fops_close(struct inode* inode, struct file* file){

    is_open--;
    module_put(THIS_MODULE);
    pr_debug("Module fops: device %s was closed\n", DEVNAME);
    return 0;
}

#ifdef COPY_MULTI_BYTES
static ssize_t fops_read(struct file* file, char* buf, size_t len, loff_t* offset) {

    int num = 0;
    if (*msg == 0) {
        pr_debug("Module fops: Msg buffer is empty!\n");
        return 0;
    }

    /* write data to buf, bytes need to be transferred from KS to US */
    while(len && *p_msg && (p_msg - msg) < msg_len) {
        put_user(*(p_msg++), buf++);
        len--;
        num++;
    }
    pr_debug("Module fops: sent %d bytes to user space\n", num);
    return num;
}

static ssize_t fops_write(struct file* file, const char* buf, size_t len, loff_t* offset){

    int num = 0;
    p_msg = msg;

    if (len > MAXMSGLEN) {
        pr_debug("Module fops: Msg buffer is too small, skipping %d bytes!\n", len - MAXMSGLEN);
    }

    /* read data from buf, bytes need to be transferred from US to KS */
    while(len && num < MAXMSGLEN) {
        get_user(*(p_msg++), buf++);
        len--;
        num++;
    }
    msg_len = num;
}
```

```

    p_msg = msg;
    pr_debug("Module fops: received %d bytes from user space\n", num);
    return num;
}

#else
static ssize_t fops_read(struct file* file, char* buf, size_t len, loff_t*
    offset) {

    int num = 0;
    if (*msg == 0) {
        pr_debug("Module fops: Msg buffer is empty!\n");
        return 0;
    }

    if (msg_len - (p_msg - msg) < len) {
        len = msg_len - (p_msg - msg);
        pr_debug("Module fops: not enough content, can read only %d bytes\n",
            len);
    }

    num = len - copy_to_user(buf, p_msg, len);
    p_msg += num;
    pr_debug("Module fops: sent %d bytes to user space\n", num);
    return num;
}

static ssize_t fops_write(struct file* file, const char* buf, size_t len,
    loff_t* offset){

    int num = 0;
    if (len > MAXMSGLEN) {
        pr_debug("Module fops: Msg buffer is too small, skipping %d bytes!\n",
            len - MAXMSGLEN);
        len = MAXMSGLEN;
    }

    num = len - copy_from_user(msg, buf, len);
    msg_len = num;
    pr_debug("Module fops: received %d bytes from user space\n", num);
    return num;
}
#endif

```

In der Funktion `fops_open()` wird über die Variable `is_open` ein Aufruf markiert und es werden weitere Versuche blockiert, sodass sich immer nur ein Gerät anmelden kann. Sollen mehrere Geräte den Treiber verwenden und soll für jede Instanz ein eigener Nachrichtenspeicher eingerichtet werden, so bietet sich zur Datenverwaltung eine verkettete Liste an. Für eine Unterscheidung der Geräte anhand ihrer Minornummer steht das Makro `iminor()` zur Verfügung. Die Inline-Funktion `try_module_get()` erhöht den Referenzzähler des Moduls und verhindert damit ein Entladen während dieses verwendet wird. Analog wird in `fops_close()` der Referenzzähler mit `module_put()` dekrementiert.

Die Funktionen `fops_read()` und `fops_write()` sind selbsterklärend. Wie bereits angesprochen, wird von verschiedenen Möglichkeiten Gebrauch gemacht, um Daten zwischen User- und Kernel-Space zu kopieren. Die `ioctl()`-Funktionen stellen eine Besonderheit der Treiberschnittstelle dar, weshalb `fops_ioctl()` im nächsten Abschnitt separat behandelt wird. Das Modul

kann allerdings jetzt schon mit einfachen Kommandozeilenprogrammen getestet werden. Nach dem Übersetzen wird der Treiber mit folgendem Befehl geladen und sollte entsprechend in der Liste `/dev/devices` auftauchen:

```
$ sudo insmod fileoperations.ko
```

Wurde eine zugehörige Gerätedatei angelegt, so lassen sich mit dem Befehl `cat` implizit die Dateioperationen `fops_open()`, `fops_read()` und `fops_close()` ausführen:

```
$ cat /dev/myfirstdevice
```

Auch wenn das Ergebnis noch nicht sehr aufsehenerregend ist, so zeigt doch `dmesg`, dass die Aufrufe tatsächlich stattfanden. Lediglich der Nachrichtenspeicher ist leer. Mit folgenden Befehlen wird eine Zeichenkette auf das Gerät geschrieben und anschließend vom Treiber wieder zurückgegeben. Die `dmesg`-Angabe bestätigt dies:

```
$ echo "Dies ist ein Test" >> /dev/myfirstdevice
$ cat /dev/myfirstdevice
$ dmesg
...
[78546.659186] Module fops: device myfirstdevice was opened from device
          with minor no 0
[78546.659197] Module fops: buffer too small, can read only 50 bytes
[78546.659200] Module fops: sent 50 bytes to user space
[78546.659206] Module fops: buffer too small, can read only 0 bytes
[78546.659207] Module fops: sent 0 bytes to user space
[78546.659210] Module fops: device myfirstdevice was closed
```

Anmerkung: Die Meldung `buffer too small` rührt daher, dass `cat` so viele Bytes wie möglich lesen möchte, im ersten `fops_read()`-Aufruf sind dies 4096. `cat` gibt sich erst zufrieden, wenn beim zweiten Aufruf als Rückgabewert 0 übermittelt wird.

### 11.5.3 Die `ioctl()`-Schnittstelle

Mit der `ioctl()`-Funktion steht eine universelle Schnittstelle zur Verfügung, mit der besondere Funktionalität bereitgestellt werden kann. So kann der Anwender zwar voraussetzen, dass jedes Gerät Daten aufnimmt (`write()`) oder liefert (`read()`). Das Setzen der Baudrate oder des Übertragungsmodus sind aber spezielle Eigenschaften eines seriellen Treibers und entsprechend als `ioctl()`-Funktionen implementiert.

Neben Zeigern auf die Strukturen `inode` und `file` bekommt die Funktion `ioctl()` ein Kommando `cmd` und ein weiteres Argument `arg` mitgeliefert. Mögliche Kommandos werden üblicherweise in der Form `IOCTL_xxx` als Makro hinterlegt. Der zugehörige Parameter ist immer vom Typ `unsigned long` und muss in der Regel in den gewünschten Zieltyp gecastet werden. Oftmals wird als Argument der Zeiger auf eine Datenstruktur mit umfangreicheren Informationen mitgeliefert. Aufgrund einer Vielzahl möglicher Kommandos bietet

sich als Gerüst der `ioctl()`-Funktion eine `switch-case`-Anweisung an (vgl. Beispiel `file_operations`):

```
static int fops_ioctl( struct inode *inode, struct file *file,
                      unsigned int cmd, unsigned long arg )
{
    switch( cmd ) {
        case IOCTL_CLRMSGBUF:
            p_msg = msg;
            msg_len = 0;
            pr_debug("Module fops: clearing message buffer\n");
            break;
        case IOCTL_SETSTDMSG:
            strncpy(msg, STDMSG, strlen(STDMSG));
            p_msg = msg;
            msg_len = strlen(STDMSG);
            pr_debug("Module fops: setting message to standard with len %d\n",
                    msg_len);
            break;
        default:
            pr_debug("unknown IOCTL 0x%x\n", cmd);
            return -EINVAL;
    }
    return 0;
}
```

In der aufgelisteten Funktion werden zwei `ioctl()`-Kommandos ausgewertet, um den Zeichenpuffer zu löschen oder diesen mit einer vorab definierten Standardnachricht zu belegen. Zu beachten ist, dass die in der Anwendung benötigten `IOCTL`-Makros in einer separaten Header-Datei definiert werden sollten, die ohne Abhängigkeiten von Kernel-Header-Dateien eingebunden werden kann. Dies ist wichtig, um die Makrodefinitionen auch in einer User-Space-Anwendung verwenden zu können. Für das aktuelle Beispiel ist das die Datei `file_operations/ioctldefs.h`.

Als Rückgabewert der Funktion wird bei erfolgreicher Ausführung 0 geliefert, ansonsten ein negativer Wert. Der Fehlercode `-EINVAL` deutet darauf hin, dass das Kommando oder die zugehörigen Parameter fehlerhaft waren.

#### 11.5.4 Verwendung von Gerätetreibern in der Anwendung

Nachdem nun die wichtigsten Dateioperationen besprochen und implementiert wurden, kann die volle Funktionalität der Schnittstelle in einer Anwendung getestet werden. Das Beispiel `file_operations_app` führt Lese- und Schreiboperationen auf dem Gerät `/dev/myfirstdevice` durch und nutzt die `ioctl`-Funktionen, um den Zeichenpuffer zu leeren oder mit einer Standardnachricht zu belegen.

Nach jeder Operation wird der Zeichenpuffer zur Überprüfung des Ergebnisses ausgelesen. Mithilfe der Funktion `printbuf()` werden nur so viele Zeichen am Bildschirm ausgegeben, wie bei einem Aufruf von `read()` empfangen wurden.

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include "../file_operations/ioctldefs.h"

char buf[MAXMSGLEN];

void printbuf(int num) {
    int i = 0;
    printf("---Device Buffer: ");
    for (i=0; i<num; i++) printf("%c",buf[i]);
    printf("\n");
}

int main() {

    int fd;
    fd = open("/dev/myfirstdevice", O_RDWR);
    if (fd < 0) {
        printf("Error opening device, fd = %d\n",fd);
        exit(0);
    }

    // test read(), write()
    int num = read(fd, buf, MAXMSGLEN);
    printbuf(num);

    sprintf(buf, "Greetings from User-Space\n");
    printf("Writing to device: %s\n", buf);
    write(fd, buf, sizeof(buf));
    num = read(fd, buf, MAXMSGLEN);
    printbuf(num);

    // test IOctls
    printf("Setting standard message via ioctl()\n");
    ioctl(fd, IOCTL_SETSTDMSG, 0);
    num = read(fd, buf, MAXMSGLEN);
    printbuf(num);

    printf("Clearing device buffer via ioctl()\n");
    ioctl(fd, IOCTL_CLRMSGBUF, 0);
    num = read(fd, buf, MAXMSGLEN);
    printbuf(num);

    close(fd);
    return 0;
}

```

Zu Beginn müssen beim Öffnen des Gerätes die Zugriffsrechte auf `O_RDWR` gesetzt werden, um Lese- und Schreibzugriffe zuzulassen. Diese Berechtigungen sind unabhängig von bereits bestehenden Zugriffsrechten im Dateisystem zu setzen. Mit den bisherigen Abschnitten ist die Grundlage für eine weiterführende Programmierung von Kernelmodulen gegeben. Ein wichtiger Punkt fehlt jedoch noch und ist für Kernelmodule von besonderem Interesse: Im nächsten Kapitel wird die direkte Kommunikation mit Hardware-Schnittstellen zum Zugriff auf reale Geräte gezeigt.

## 11.6 Hardware-Zugriff

Ein Hardware-Zugriff kann grundsätzlich auf verschiedenen Ebenen geschehen. In Abschnitt 11.6.1 wird zunächst die Verwendung von IO-Ports und IO-Speicher beschrieben. Diese Variante wird benötigt, falls selbst entwickelte Hardware bspw. an einem ISA-, PCI-Bus oder Parallelport verwendet werden soll. Werden herkömmliche Schnittstellen wie RS232 verwendet, so kann für eine Funktionserweiterung oder -anpassung auf anderen Treibern aufgebaut werden. In Abschnitt 11.6.2 wird das Kommunikationsprotokoll für eine serielle Relaisplatine in ein Kernelmodul integriert (vgl. Kapitel 7).

Grundsätzlich ist die Ansteuerung der Hardware-Komponenten und die Absicherung gegen eine fehlerhafte Anwendung auf einem eingebetteten System weit weniger kritisch, als auf einem Mehrbenutzer-PC-System. Dies bedeutet nicht, dass mit den Ressourcen fahrlässig umgegangen werden darf. Im Regelfall ist aber auf einem eingebetteten System bekannt, welche Dienste aktiviert sind, welche Schnittstellen von bestimmten Anwendungen genutzt werden und was für Geräte dort angeschlossen sind. Eine schnelle und gut funktionierende Lösung ist damit oftmals der vollständig abgesicherten Variante vorzuziehen.

### 11.6.1 Zugriff über IO-Ports und IO-Speicher

Ein wichtiger Teil der Systemressourcen sind IO-Ports und IO-Speicher. Diese werden oftmals unter dem Begriff IO-Bereiche zusammengefasst. Bevor ein Gerät über einen IO-Bereich angesprochen werden kann, müssen die zugehörigen Bereiche identifiziert werden. Diese Aufgabe übernehmen Treiber, die beim Laden nach typischen Eigenschaften bestimmter Geräte oder Komponenten suchen. Beim ISA-Bus war die Identifikation besonders kritisch, da hier auf gut Glück einzelne Speicherbereiche beschrieben und wieder gelesen wurden. Dieser Sachverhalt hat sich mit Verwendung des PCI-Busses aber verbessert. Findet ein Treiber ein Gerät unter einer bestimmten Ressource, so wird diese alloziert und damit eine Kollision mit anderen Treibern vermieden.

In den Dateien `/proc/ioports` und `/proc/iomem` werden die von Treibern registrierten IO-Bereiche hinterlegt. Ob der IO-Bereich einer Schnittstelle den Ports oder dem Speicher zugeordnet wird, hängt sowohl von der Prozessorarchitektur ab, als auch von der Frage, ob für IO-Ports ein eigener Adressraum zur Verfügung steht. Bei vielen Embedded-Prozessoren ist dies nicht der Fall, hier sind in `/proc/ioports` kaum Einträge zu finden. Stattdessen sind diese Ports Teil des normalen Adressraumes, und der Zugriff erfolgt mit Speicherzugriffsbefehlen. Man spricht in diesem Zusammenhang auch von *Memory-Mapped-IO*. In x86-Architekturen steht üblicherweise ein eigener IO-Adressraum zur Verfügung, sodass eine typische Auflistung der IO-Ports auf einem PC folgendermaßen aussieht:

```
$ cat /proc/ioports
...
0050-0053 : timer1
0060-006f : keyboard
...
0378-037a : parport0
03c0-03df : vga+
03f6-03f6 : 0000:00:1f.1
03f8-03ff : serial
0400-047f : 0000:00:1f.0
...
0500-053f : 0000:00:1f.0
0680-06ff : smsc47m1
0778-077a : parport0
0cf8-0cff : PCI conf1
```

Der Portbereich **0378-037a** wird von einem Parallelport-Treiber verwendet, **03f8-03ff** von der seriellen Schnittstelle. Zur Verwaltung der Port-Ressourcen stehen drei in `linux/ioport.h` definierte Funktionen zur Verfügung:

```
int check_region(unsigned long from, unsigned long len);
```

Diese Funktion überprüft, ob der IO-Bereich ab Startadresse `from` mit einer Länge von `len` Bytes zur Verfügung steht. Der Speicherbereich wird noch nicht reserviert. Wenn der Bereich frei ist, dann wird 0 zurückgegeben, sonst ein Fehlercode.

```
struct resource* request_region(unsigned long from, unsigned
    long len, const char *name);
```

Die Funktion reserviert einen IO-Bereich der Länge `len` ab der Adresse `from` unter dem Namen `name`. War die Aktion erfolgreich, so wird die Speicheradresse zurückgegeben, sonst 0.

```
void release_region(unsigned long from, unsigned long length);
```

Hiermit wird ein zuvor allozierter IO-Bereich wieder freigegeben.

Ein Beispiel ist die Zweckentfremdung des Parallelports: Hier wäre die korrekte Lösung die Reservierung des Bereiches in einem eigenen Kernelmodul nachdem das Modul `parport_pc` entladen wurde. Damit fällt jedoch die komplette Organisation der Schnittstelle in die Verantwortung des eigenen Moduls. Dies bedeutet, dass die Hardware-Abstraktion, welche im Modul `parport_pc` enthalten ist, selbst programmiert oder übernommen werden muss. Konkret folgt daraus, dass ohne besagtes Modul das Datenregister der parallelen Schnittstelle nicht mehr direkt angesprochen werden kann.

Für eine rasche Lösung werden lediglich die nicht benötigten Module `lp` und `ppdev` aus dem Kernel entfernt und das Modul `parport_pc` beibehalten:

```
$ sudo rmmod lp ppdev
```

Dies bewirkt, dass ein Zugriff auf die Register der parallelen Schnittstelle in herkömmlicher Weise möglich ist und Daten mit folgenden Befehlen auf die Register geschrieben bzw. davon gelesen werden können (vgl. Abschnitt 7.4):



```
outb(0xFF,0x378);    // schreibt 0xFF auf das Datenregister
char val = inb(0x37a); // liest ein Byte vom Kontrollregister
```

Allerdings ist die Ressource nach wie vor belegt, sodass auf eine Allokation im eigenen Modul verzichtet werden muss. In Beispiel `blink` wird die parallele Schnittstelle zur Ausgabe von Blinksignalen verwendet.<sup>8</sup>

Nach dem Übersetzen und Laden des Moduls wird eine zugehörige Gerätedatei als Einstiegspunkt erstellt:

```
$ cd <embedded-linux-dir>/examples/kernel-modules/blink/
$ make
$ sudo insmod ./blink.ko
$ sudo mknod /dev/blink -m 666 c 60 0
```

Nun lässt sich die Blinkfrequenz mit dem Befehl `echo` setzen und mit `cat` abfragen:

```
$ echo 5 > /dev/blink
$ cat /dev/blink
5
```

Das zusätzliche Entfernen des Gerätes `/dev/parport0` gibt noch keine Garantie dafür, dass der Treiber nicht von anderer Stelle verwendet wird. Es minimiert aber das Risiko eines versehentlichen Zugriffs durch einen Druckertreiber. Wird ein eigener Treiber für eine ISA- oder PC104-Karte implementiert, so ist der IO-Adressbereich der Karte zunächst per Jumper auf einen freien Bereich einzustellen, bevor dieser im Treiber mit `request_region()` zugeordnet werden kann.

Der im Beispiel verwendete Kernel-Timer eignet sich gut, um Funktionen zu einem bestimmten Zeitpunkt auszuführen (hier: das Invertieren der Portleitungen). Dies ist neben der einfachen Verwendung der größte Vorteil gegenüber Task-Schlangen. Kernel-Timer sind nicht zyklisch und müssen deshalb in der aufgerufenen Funktion erneut gesetzt werden. Weitere Informationen zu Kernel-Timern können unter <http://www.oreilly.de/german/freebooks/linuxdrive2ger/flowtimers.html> nachgelesen werden.

Wie bereits angesprochen, so steht für Embedded-CPU's oft kein eigener Adressbereich für IO-Ports zur Verfügung. Bei einer NSLU2 bspw. sind die IO-Adressen Teil des Adressraums. Die Datei `/proc/iomem` macht dies deutlich:

```
$ cat /proc/iomem
00000000-01ffffff : System RAM
0001f000-0020636f : Kernel text
...
```

<sup>8</sup> Vgl. die Testschaltung hierzu in Abbildung 7.3. Für eine detaillierte Übersicht über die Pin- und Registerbelegung der parallelen Schnittstelle sei auf Abschnitt 7.3 und <http://www.microfux.de/Fundgrube/Hardware/PrinterPort/printerport.htm> verwiesen.

```
c8000000-c8000fff : serial8250.0
c8000000-c800001f : serial
...
```

IO-Speicher wird auf ähnliche Weise beantragt und reserviert wie IO-Ports. Es stehen dafür die Funktionen `check_mem_region()`, `request_mem_region()` und `release_mem_region()` zur Verfügung.

### 11.6.2 Zugriff über das Dateisystem

Bestimmte Zugriffe sollten im Kernel auf jeden Fall unterlassen werden. Hierzu gehört auch, auf Dateien oder Gerätedateien im User-Space zuzugreifen. Dies hängt in erster Linie damit zusammen, dass zwischen Kernel-Space und User-Space in solch einem Fall Vereinbarungen über Namen und Ort der Datei getroffen werden müssen. Diese Vereinbarungen sind in der Kernel-Entwickler-Gemeinde nicht gern gesehen. Verschwindet bspw. die Datei aus irgendeinem Grund, so wird dies für das Kernelmodul problematisch. Für Gerätedateien, die in der Regel nicht verschoben werden, mag der Sachverhalt etwas unkritischer sein. Es geht aber gerade bei Kernelfragen oft um die saubere Einhaltung bestimmter Prinzipien und in vielen Foren wird davon abgeraten. Lässt sich der Programmierer davon nicht beeindrucken und wird das Kernelmodul gewissenhaft implementiert und abgesichert, so kann diese Variante zu einer pragmatischen Lösung führen. Ein großer Vorteil ist, dass die Ansteuerungsroutinen, die im User-Space für eine Schnittstelle entwickelt und intensiv getestet wurden, fast identisch in das Kernelmodul übernommen werden können. Der Anwender spart sich im Fall der seriellen Schnittstelle bspw. die Programmierung eines eigenen TTY-Treibers (vgl. Kapitel 7 zu TTY). Andererseits muss sichergestellt werden, dass die im User-Space nach wie vor vorhandene Schnittstelle nur vom Kernelmodul selbst und von keinem anderen Dienst verwendet wird. Auf einem eingebetteten System ist dies jedoch ein akzeptabler und durchführbarer Kompromiss.

Die serielle Relaiskarte (vgl. Tabelle E.1 im Anhang) wurde bereits in Kapitel 7 vorgestellt und ist entsprechend schon bekannt. Das relativ einfache Kommunikationsprotokoll soll im Beispiel `relaiscard` in einen Kerneltreiber integriert werden, welcher auf die Gerätedatei im User-Space zugreift und seinerseits acht einzelne Gerätedateien zur Ansteuerung der Relais bereitstellt.

Ein Aufsplitten der Relaiskarte in einzelne Geräte bedeutet den Vorteil, dass verschiedene Anwendungen unabhängig voneinander auf die gleiche Ressource zugreifen können und diese vom Kernelmodul zentral verwaltet wird. Ein Bash-Skript beispielsweise oder ein Cronjob wären in der Lage, die zugeordneten Relais separat anzusteuern ohne von den anderen Relaiszuständen zu wissen und ohne diese womöglich zu verändern.

Bei Betrachtung des Beispiels **relaiscard** fällt im Gegensatz zu den bisherigen Modulen auf, dass ein Zugriff von bis zu acht Geräten koordiniert werden muss. Diese werden anhand ihrer Minornummer unterschieden. Weiterhin werden Zustände für *geöffnet* oder *neue Leseanforderung* in den Feldern `is_open[8]` bzw. `new_read[8]` abgelegt. Die Thematik des wiederholten Lesezugriffs bei einem `cat`-Aufruf (bis als Rückgabewert 0 folgt) ist aus Abschnitt 11.5.2 bekannt. Über `is_open[num]` wird die Rückgabe entweder des aktuellen Zustandes oder einer abschließenden 0 koordiniert.

Ähnlich wie im User-Space ist das Öffnen einer Gerätedatei mit Rückgabe eines Filedescriptors auch im Kernel-Space möglich:

```
fd_p = filp_open(src, 0_RDWR, 0);
if (fd_p < 0) {
    pr_debug("Module relaiscard: Error opening %s\n",src);
    return -1;
}
```

Ein Schreibzugriff erfolgt unter Angabe des Filedescriptors, der Speicheradresse mit den bereitliegenden Daten (im Kernel-Space) und der Anzahl zu schreibender Daten-Bytes:

```
int ret = fd_p->f_op->write(fd_p,buf,4,0);
```

Der Kernel erwartet, dass der Aufruf der Dateioperation aus dem User-Space stattfindet, dies ist innerhalb des Kernelmoduls schwer möglich. Die vom Kernel durchgeführte Überprüfung des Zeigers schlägt mit der Meldung **EFAULT** fehl. Durch eine Änderung der Adressbeschränkungen des aktuellen Prozesses kann dieses Problem umgangen werden. Mit den Befehlen `get_fs()` und `set_fs()` werden diese Grenzen abgefragt bzw. gesetzt. Folgender Code-Abschnitt hinterlegt die aktuelle Prozessbeschränkung in `fs` und erweitert anschließend die Adresslimitierung auf den gesamten Kernel-Space. Damit werden alle Pointer akzeptiert, die aus dem Kernel kommen. Nach Ausführung der User-Space-Dateioperation wird wieder der ursprüngliche Addressraum hergestellt:

```
static mm_segment_t fs = get_fs();
set_fs(KERNEL_DS);
// do user space call
set_fs(fs);
```

Auf die einzelnen seriellen Ansteuerungsroutinen soll an dieser Stelle nicht näher eingegangen werden, sie ähneln denen der Klasse **Relaisboard** (vgl. Abschnitt 7.2.3). Der einzige Unterschied ist die Verwendung von sog. *Spinlocks*, um die Schreib- und Lesezugriffe auf den seriellen Bus atomar bzw. unteilbar zu halten. Spinlocks erzeugen ununterbrechbare Bereiche und können verwendet werden, um gemeinsam genutzte Ressourcen vor konkurrierenden Prozessen zu schützen. Spinlocks warten aktiv (*spinning*), bis das kritische Code-Segment freigegeben ist. Dieses Verfahren ist auf Mehrprozessor-Rechnern unbedingt notwendig, da Kernel-Thread und *Interrupt-Service-Routine (ISR)* auf

unterschiedlichen Prozessoren laufen können und ein Kernel-Thread eine gerade parallel ablaufende ISR nicht sperren kann. Obwohl Spinlocks nur auf Multi-Prozessor-Systemen notwendig sind, ist eine Verwendung im Quellcode möglich. Durch entsprechende Makros werden auf Einprozessormaschinen stattdessen Interruptsperrern verwendet. Im Gegensatz zu einer Sperre mittels `spin_lock()` verhindert eine Absicherung über `spin_lock_irq()`, dass die Ausführung durch eine ISR unterbrochen werden kann. Im vorliegenden Fall reicht `spin_lock()` aus. Grundsätzlich ist jedoch Vorsicht geboten, um keine *Deadlocks* zu erzeugen. Diese könnten auftreten, falls während einer Blockade in eine ISR-Routine gesprungen wird, die ihrerseits auf eine Freigabe des Spinlocks wartet.<sup>9</sup> Bevor das Modul übersetzt und geladen wird, sollten zunächst die einzelnen Relais-Gerätedateien erzeugt werden. Weiterhin ist es sinnvoll, die Parameter der seriellen Schnittstelle bereits im User-Space korrekt zu setzen. Für beide Aktionen stehen Bash-Skripte im Beispielerverzeichnis zur Verfügung:

```
$ sudo ./make_devices
$ ./set_stty
```

Wird das Modul geladen, so können einzelne Relais über `echo` geschaltet oder die Zustände über `cat` abgefragt werden:

```
$ sudo insmod ./relaiscard.ko device="/dev/ttyS0"
$ echo 1 > /dev/relais4
$ cat /dev/relais4
1
```

Ohne Angabe der seriellen Gerätedatei kann das Modul nicht geladen werden. Das Anwendungsbeispiel `relaiscard.app` zeigt den Zugriff auf ein einzelnes Relais aus einer Anwendung heraus. Werden mehrere Anwendungen parallel gestartet, so übernimmt das Kernelmodul die zentrale Koordination.

```
$ ./relais_app /dev/relais1 &
$ bg
$ ./relais_app /dev/relais2 &
$ bg
$ ./relais_app /dev/relais3 &
```

Die Relais schalten auch bei deaktivierten Spinlocks auf einem Einprozessorsystem korrekt. Findet die Ausführung auf einem Mehrprozessorsystem statt, so können Anwendungen auch auf mehrere Kerne verteilt werden. Durch die Verwendung von Spinlocks ist man aber auf der sicheren Seite.

Das vorliegende Kapitel konnte nur einen kleinen Blick in die Welt der Kernelprogrammierung geben. Nicht umsonst sind diesem komplexen Thema bereits viele Bücher gewidmet. Für Leser, deren Interesse an der der Thematik geweckt wurde, sei als weiterführende Literatur [Quade 06] empfohlen.

<sup>9</sup> Weiterführende Informationen zur Absicherung kritischer Abschnitte können unter <http://ezs.kr.hs-niederrhein.de/TreiberBuch/html/sec.csschutz.html> nachgelesen werden.

## Multithreading

*Autor: Daniel Jagszent*

### 12.1 Einführung

Im Kapitel 1 wurde der Prozess-Scheduler als ein Vorteil von Betriebssystemen vorgestellt, der auch für eingebettete Systeme von Nutzen sein kann, da hiermit mehrere Tasks quasi-parallel<sup>1</sup> ausgeführt werden können. Dieses sog. Multitasking ermöglicht es auch eingebetteten Systemen auf unkomplizierte Weise mehrere Aufgaben auszuführen.

Das vorliegende Kapitel führt Threads<sup>2</sup> als eine weitere Abstraktionsstufe ein. Sie haben längst Einzug in moderne Betriebssysteme und so auch in Linux gehalten. Für die effiziente Entwicklung komplexerer Programme sind Threads zu einem integralen Standardhilfsmittel geworden. Sie werden z. B. eingesetzt, um die Bedienoberfläche eines Programms von der Programmlogik zu trennen, rechenaufwändige Algorithmen im Hintergrund laufen zu lassen, oder um nebenläufige Netzwerkkommunikation zu ermöglichen (vgl. hierzu auch das Kapitel 13).

Der folgende Abschnitt erklärt zuerst einige Grundlagen für jedwede Art der nebenläufigen Programmierung, um im Folgenden näher auf die Thread-Programmierung am Beispiel der Posix-Schnittstelle einzugehen. Nach diesen Grundlagen wird eine schlanke C++-Schnittstelle vorgestellt, die in den folgenden Kapiteln für die nebenläufige Programmierung verwendet wird. Abschließend wird ein Anwendungsbeispiel betrachtet, das die Vorteile der Threads ihm Rahmen eingebetteter Systeme demonstriert.

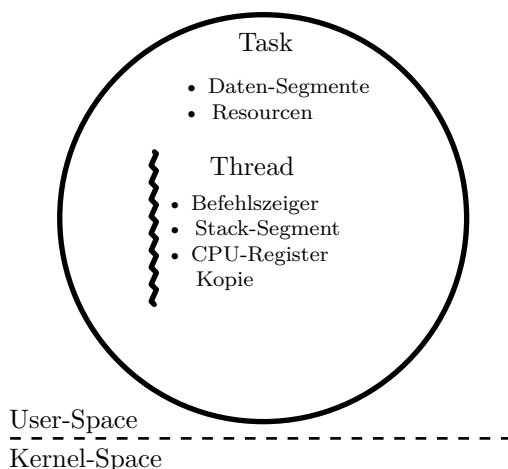
---

<sup>1</sup> Nur auf Mehrprozessorsystemen ist wirkliche Parallelität möglich.

<sup>2</sup> Engl. für *Faden*; Kurzform von *thread of execution*, was als *Ausführungsfaden* übersetzt werden kann.

Die im Rahmen dieses Kapitels vorgestellten Beispielprogramme sind im Verzeichnis `<embedded-linux-dir>/examples/threads` zu finden.

## 12.2 Grundlagen



**Abb. 12.1.** Ein Task kann mehrere Threads enthalten. Ein Task kapselt die Ressourcen, ein Thread einen Ausführungsfaden.

Auch in diesem Kapitel wollen wir, genauso wie in Kapitel 1, an der grundsätzlichen Notation eines Tasks als die von anderen Tasks abgeschirmte Ausführung eines Programmcodes im User-Space festhalten. Durch diese Abschirmung, die das Betriebssystem zur Verfügung stellt, können die einzelnen Tasks vereinfacht annehmen, dass sie der einzige ausgeführte Prozess sind. Gemeinsam verwendbare Ressourcen wie der Arbeitsspeicher oder die CPU werden durch das Betriebssystem unter allen laufenden Tasks aufgeteilt. Nicht teilbare Ressourcen wie z. B. der Parallelport werden vom Betriebssystem einem Task exklusiv zugesprochen.

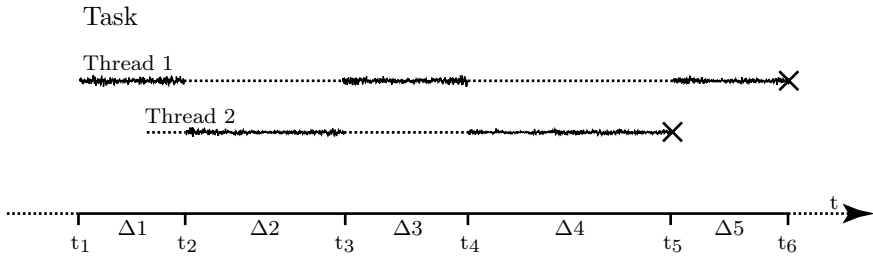
Wie bereits kurz in Kapitel 11 erwähnt, gibt es außer der Task-Abstraktion in Linux noch die feingranularere Thread-Abstraktion:

*Ein Thread ist innerhalb des Tasks die Abstraktion der Ausführung. Ein Thread besitzt einen Befehlszeiger<sup>3</sup>, eine Kopie der CPU-Register zur Speicherung sei-*

<sup>3</sup> Ein Zeiger, der auf den nächsten auszuführenden Befehl zeigt.

nes lokalen Zustandes und ein Stack-Segment<sup>4</sup> zur Speicherung seiner Aufrufhistorie.

Ein Task besitzt immer mindestens einen Thread, der automatisch bei Programmstart erstellt wird und dessen Lebensspanne die Lebensspanne des kompletten Tasks definiert. In einem C/C++-Programm repräsentiert die `main()`-Funktion diesen speziellen Thread.



**Abb. 12.2.** Ein mögliches Prozess-Scheduling für zwei Threads eines Tasks. Die Lebensspanne des ersten Threads bestimmt die Lebensspanne des Tasks.

Der Prozess-Scheduler teilt nicht mehr den Tasks, sondern den einzelnen Threads Prozessorzeit zu. Abbildung 12.2 zeigt z.B. einen Task mit zwei Threads und einen möglichen Prozessablauf für diese Threads: Zum Zeitpunkt  $t_1$  wird der Task und mit ihm sein Haupt-Thread *Thread 1* erstellt. Zur Zeitspanne  $\Delta_1$  bekommt der erste Thread in diesem Beispiel auch den Prozessor zugeordnet. In dieser Zeitspanne erzeugt er den zweiten Thread des Programms, *Thread 2*. Der neue Thread wird aber nicht sofort aktiviert. Erst ab dem Zeitpunkt  $t_2$  bekommt er vom Prozess-Scheduler die gemeinsame Ressource *Prozessor* zugewiesen. Nun folgen einige Thread-Wechsel, bis zum Zeitpunkt  $t_5$  *Thread 2* beendet wird. Der Haupt-Thread ist nun wieder der einzige Thread, bis auch er sich zum Zeitpunkt  $t_6$  beendet und damit das Ende des Tasks bewirkt. Der Prozess-Scheduler kann natürlich nicht nur zwischen Threads des gleichen Tasks wechseln, sondern er ist auch in der Lage, Threads aus unterschiedlichen Tasks zu aktivieren. Das Wechseln zwischen zwei Threads des gleichen Tasks ist dabei schneller, da der Prozess-Scheduler hierbei nur die folgenden zwei Aufgaben bearbeiten muss:

1. Er sichert die CPU-Register und schreibt sie in die Kopie des zuvor laufenden Threads. Eventuell müssen noch der Befehlszeiger und der Stackzeiger gespeichert werden. In der Regel befinden sich diese beiden Zeiger allerdings in speziellen CPU-Registern, was ein gesondertes Speichern dieser Zeiger überflüssig macht.

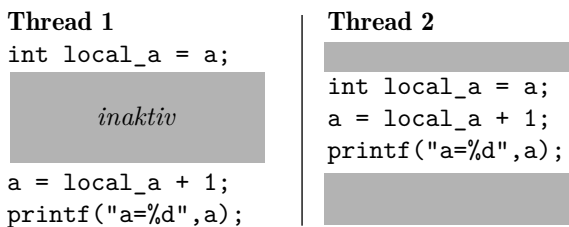
<sup>4</sup> Ein *last-in-first-out* Speicher, auf dem die lokalen Variablen bei Unterprozeduraufrufen zwischengespeichert werden.

2. Er lädt die gespeicherte Kopie der CPU-Register und eventuell Kopien der Befehls- und Stackzeiger des zu aktivierenden Threads.

Im Gegensatz hierzu ist das Wechseln zwischen zwei Threads, die sich in verschiedenen Tasks befinden um ein Vielfaches aufwändiger, da u. a. noch der Adressraum gewechselt und der Cache gelöscht werden muss.

Da die Threads eines Tasks sich einen Adressraum und gemeinsame Ressourcen teilen, können sie viel enger kooperieren als zwei unterschiedliche Tasks. Beispielsweise könnte ein GUI-Thread und ein Thread zum Einlesen von Kamerabildern auf den gleichen Speicherbereich zugreifen, was es der GUI einfach macht, das Bild der Kamera auf dem Bildschirm anzuzeigen. Diese enge Verzahnung und die fehlenden Grenzen bringen aber auch Probleme mit sich. So könnte ein fehlerhafter Thread wahllos Speicherbereiche überschreiben und somit kritische Fehler in den Threads des gleichen Tasks auslösen. Da der Programmierer im Allgemeinen die volle Kontrolle darüber hat, welche Threads in seiner Anwendung ausgeführt werden, ist dies glücklicherweise in einer Standardanwendung kein Problem. Das folgende allgemeine Problem der Thread-Programmierung betrifft allerdings alle Arten von Anwendungen:

Immer wenn zwei oder mehr Threads auf die gleiche Variable oder Ressource zugreifen, müssen sich die beteiligten Threads gegenseitig absprechen, damit es nicht zu Konflikten kommt. Das folgende Beispiel, in dem zwei Threads sich die globale Integer-Variable `a` teilen, soll einen solchen Konflikt verdeutlichen. Jeder Thread kopiert die globale Variable in eine für den Thread lokale Variable und schreibt in die globale Variable einen um 1 erhöhten Wert. Nachdem beide Threads ausgeführt wurden, sollte die globale Variable entsprechend den Wert 2 enthalten, wenn sie mit 0 initialisiert wurde.



**Abb. 12.3.** Eine *Race Condition*: Ausgegeben wird zweimal `a=1`, da das Ergebnis von *Thread 2* durch *Thread 1* überschrieben wird. Vorbedingung: `a=0`.

In ungünstigen Fällen ist aber das in Abbildung 12.3 dargestellte Prozess-Scheduling der beiden Threads denkbar, mit welchem dies nicht gegeben ist:

Nachdem *Thread 1* die lokale Kopie der Variablen erstellt hat, wird *Thread 2* aktiviert. Auch dieser Thread holt sich eine lokale Kopie der globalen Varia-



blen, erhöht diese um 1 und speichert den Wert wieder in der globalen Variablen. `a` enthält damit den Wert 1. Nun wird wieder *Thread 1* aktiviert und er überschreibt die globale Variable mit seiner um 1 erhöhten lokalen Kopie. Nach seiner Ausführung enthält die globale Variable `a` den Wert 1, und das Ergebnis des zweiten Threads ist verloren.

Solche Situationen, in denen zwei oder mehr Threads einen gemeinsam genutzten Speicher konkurrierend verwenden und das Endergebnis von der Ausführungsreihenfolge der beteiligten Threads abhängt, werden *Race Conditions* genannt. Race Conditions treten nur selten auf, müssen aber trotzdem beachtet werden, um ein korrektes Ergebnis zu garantieren. Nach Murphys Gesetz<sup>5</sup> tritt ein möglicher Fehlerfall auch stets ein.

Ein Programm oder Programmteil heißt *thread-sicher* oder auch *thread-safe*, wenn sichergestellt ist, dass keine Race Conditions auftreten können. Dann hängt auch das Endergebnis nicht davon ab, in welcher Reihenfolge die beteiligten Threads ausgeführt werden und an welcher Stelle ein Thread unterbrochen wird.

Bereiche, in denen auf gemeinsame Ressourcen zugegriffen wird, heißen *kritische Abschnitte* oder *kritische Bereiche*. Race Conditions können vermieden werden, wenn der Eintritt in solche kritischen Abschnitte serialisiert wird. Damit wird sichergestellt, dass jeweils nur ein Thread in einem kritischen Abschnitt ist. Die Fachliteratur wie bspw. [Tanenbaum 02] nennt hierfür mehrere Möglichkeiten. Hier soll nur auf eine Möglichkeit genauer eingegangen werden: Der gegenseitige Ausschluss durch *Mutexe*.

### *Mutexe*

Eine Mutex<sup>6</sup> ist eine Variable mit zwei möglichen Zuständen: Gesperrt oder nicht gesperrt. Auf dieser Variablen sind zwei Operationen `mutex_lock()` und `mutex_unlock()` definiert. Mithilfe einer Mutex lassen sich kritische Abschnitte absichern: Ein Thread ruft vor dem Betreten seines kritischen Abschnitts `mutex_lock()` auf. Diese Funktion wird den Thread so lange blockieren, bis kein anderer Thread in einem zugehörigen kritischen Abschnitt ist und somit die Mutex gesperrt hat. Nach dem Aufruf von `mutex_lock()` *besitzt* der Thread die Mutex. Jeder andere Thread, der nun seinen kritischen Abschnitt betreten will und deshalb `mutex_lock()` auf die gleiche Mutex aufruft, wird so lange blockiert, bis der Besitzer der Mutex sie wieder mit `mutex_unlock()` entsperrt.

---

<sup>5</sup> Alles, was schiefgehen kann, wird auch schiefgehen.

<sup>6</sup> Kofferwort vom englischen *mutual exclusion* – gegenseitiger Ausschluss.

Die Vorteile der Mutexe gegenüber anderen Verfahren zum gegenseitigen Ausschluss kritischer Abschnitte sind, dass durch sie aktives Warten<sup>7</sup> vermieden wird und dass durch die Einfachheit dieses Verfahrens in der Regel eine schnelle und effiziente Implementierung möglich ist.

Zum einen ist für eine Mutex keine umfangreiche Datenstruktur erforderlich, da ein einzelnes Bit für die Darstellung ausreicht (gesperrt/nicht gesperrt). Zum anderen kann auf den meisten CPU-Architekturen für ihre Implementierung auf die TSL-Assembler-Instruktion<sup>8</sup> zurückgegriffen werden. Die nicht unterbrechbare TSL-Instruktion liest den Wert einer Speicheradresse in ein Register und schreibt 1 als neuen Wert an diese Adresse. In `mutex_lock()` kann entsprechend mithilfe der TSL-Instruktion eine 1 in die Mutex geschrieben und danach ermittelt werden, ob auch vorher eine 1 dort stand. Ist dies der Fall, so stellt der aktuelle Thread seine in diesem Zeitschlitz noch zur Verfügung stehende Rechenzeit anderen Threads zur Verfügung und versucht, die Mutex in seinem nächsten Zeitschlitz zu bekommen. `mutex_unlock()` speichert eine 0 in die Mutex-Variable. Beide Funktionen, `mutex_lock()` und `mutex_unlock()`, können mit wenigen Assembler-Instruktionen im User-Space implementiert werden; ein zeitaufwändiges Wechseln in den Kernel-Space kann entfallen.

Eine Mutex kann als Basis für komplexere Synchronisationsmechanismen verwendet werden. Sie kann z. B. die Datenstrukturen einer *Semaphore* absichern. Eine Semaphore ist ein häufig verwendeter Synchronisationsmechanismus mit einer komplexeren Datenstruktur, einer Zählvariablen und einer Warteliste, auf welcher die zwei Operationen `up()` und `down()` definiert sind. Auf Semaphoren soll hier nicht weiter eingegangen werden, da sie im Folgenden nicht verwendet werden.

Werden mehrere Ressourcen durch Mutexe abgesichert und mehrere Threads greifen parallel auf diese Ressourcen zu, so kann es zu *Verklemmungen*<sup>9</sup> kommen: Eine Menge von Threads befindet sich in einem Verklemmungszustand, wenn jeder Thread dieser Menge auf eine Mutex wartet, die ein anderer Thread aus der Menge reserviert hat.

### *Zustandsvariablen*

Mit Mutexen lässt sich zwar der gegenseitige Ausschluss realisieren, die kooperative Zusammenarbeit aber von zwei oder mehr Threads ist mit ihnen nicht einfach möglich. Unter kooperativer Zusammenarbeit versteht man das bewusste Zusammenarbeiten von mehreren Threads, um ein gemeinsames Ziel

<sup>7</sup> Beim *aktiven Warten* wird die Wartebedingung ständig ohne Unterbrechung überprüft.

<sup>8</sup> *Test and Set Lock*, bei Intels i386-Architektur ist BTS (*Bit Test Set*) mit LOCK-Prefix eine vergleichbare Instruktion.

<sup>9</sup> Engl. *deadlock*. Auch im Deutschen werden Verklemmungen oft Deadlock genannt.

zu erreichen. Um eine kooperative Zusammenarbeit zu erleichtern, werden an dieser Stelle Zustandsvariablen eingeführt.<sup>10</sup>

Eine Zustandsvariable besteht aus eine Liste von Threads und drei Operationen, die auf dieser Liste arbeiten:

- `wait()` fügt den aufrufenden Thread zur Thread-Liste hinzu und markiert den Thread als blockiert.
- Wenn die Thread-Liste nicht leer ist, so entfernt `signal()` einen<sup>11</sup> Thread aus der Liste und entsperrt diesen, sodass er vom Prozess-Scheduler wieder berücksichtigt wird.
- Die Operation `signalAll()` leert die komplette Warteliste und entsperret alle wartenden Threads.

Mithilfe der Warteliste und dieser drei Operationen lassen sich Ereignisse wie *Es liegt ein Ergebnis vor* oder *Warte auf nächsten Job* zwischen Threads übermitteln. Der Thread, der auf ein Ereignis wartet, ruft `wait()` auf der Zustandsvariablen für dieses Ereignis auf und wird damit so lange blockiert, bis ein anderer Thread ihn mit `signal()` oder `signalAll()` wieder aufweckt und somit signalisiert, dass das Ereignis eingetreten ist. Sollte das Ereignis allerdings bereits vor dem Aufruf von `wait()` eingetreten sein, der signalisierende Thread entsprechend bereits `signal()` auf die Zustandsvariable aufgerufen haben, dann ist das Ereignis verloren. Der wartende Thread wird so lange blockiert, bis ein weiteres Signal auf der Zustandsvariablen ihn wieder aufweckt.

## 12.3 Posix-Schnittstelle

Das Portable Operating System Interface (Posix) ist ein von der IEEE und der OpenGroup entwickelter Standard, um die Programmierung unterschiedlicher Unix-Betriebssysteme auf Quellcodeebene zu vereinheitlichen. Der Posix-Standard enthält mehrere Erweiterungen, unter anderem auch eine Erweiterung für Threads (POSIX.4a, im Englischen oft *Pthread* genannt) auf die im Folgenden näher eingegangen werden soll, da sie die Standard-Schnittstelle zur Thread-Programmierung unter Linux darstellt. Jede andere, komfortablere Thread-Schnittstelle setzt hier auf.

Nachfolgend werden einige der im Posix-Thread-Standard definierten Funktionen vorgestellt. Sie befinden sich alle in der Standard-Header-Datei `pthread.h`.

<sup>10</sup> Engl. *condition variables*; zuerst entwickelt für *Monitore*, eine andere Synchronisationsmethode, die auf Syntaxerweiterungen der Programmiersprache setzt und z. B. in Java implementiert ist.

<sup>11</sup> In aller Regel ist dies der ersten Thread in der Liste.

Eine vollständige Liste aller in dieser Header-Datei definierten Funktionen und ihre Beschreibung finden sich z.B. unter: <http://opengroup.org/onlinepubs/007908799/xsh/pthread.h.html>.

### 12.3.1 Thread-Funktionen

```
int pthread_create(pthread_t *thread, const pthread_attr_t *
attr, void *(*start_routine)(void*), void *arg);
```

Mithilfe von `pthread_create()` wird ein neuer Thread erzeugt und auch sofort gestartet. Der Befehlszeiger des neuen Threads wird auf den Anfang der als dritter Parameter übergebenen Funktion gesetzt. Der neue Thread wird automatisch beendet, wenn diese Funktion wieder verlassen wird. Der erste Parameter, `thread`, ist ein Zeiger auf eine Struktur, die von `pthread_create()` gefüllt wird und die nach dem Aufruf den neu erstellten Thread identifiziert. Der zweite Parameter braucht im Standardfall nicht beachtet zu werden und kann auf `NULL` gesetzt werden. Er ermöglicht die Erstellung von Threads mit erweiterten Optionen. Der Parameter `arg` wird der Thread-Funktion des neu erstellten Threads als Argument übergeben.

Das folgende Beispiel zeigt, wie man mit `pthread_create()` einen neuen Thread mit Standardparametern erstellt und ihm eine Zeichenkette als Argument übergibt:

```
#include <pthread.h>
#include <stdio.h>

void* thread_main(void* arg) {
    printf("%s, hier Thread!\n", (char*)arg);
}

const char* data = "Hallo Welt";

int main() {
    pthread_t th;
    pthread_create(&th, NULL, thread_main, data);
}
```

Wie jedes Programm, das Posix-Thread-Funktionen benutzt, so muss auch dieses Beispiel mit der `pthread`-Bibliothek gebunden werden. Folgender Aufruf des GCC-Compilers führt dies aus:

```
$ gcc -lpthread -o threads_basic main.c
```

Führt man das so übersetzte Beispielprogramm aus, so erscheint überraschenderweise keine Ausgabe. Zu erwarten wäre der Text "Hallo Welt, hier Thread!". Um diesen Umstand zu verstehen, muss man wissen, dass die Hauptroutine eine Sonderstellung einnimmt, denn die Lebensdauer dieses Threads ist gleichbedeutend mit der Lebensdauer des ganzen Tasks. Mit diesem Wissen wird klar, warum das Beispielprogramm keine

Ausgabe anzeigt: Der neu erzeugte Thread kommt nicht dazu, die Ausgabe zu bearbeiten, weil der komplette Task vorher beendet wird. Fügt man nach dem `pthread_create()` weitere Befehle wie z. B. ein `sleep(3)`; ein, so erscheint die Ausgabe. Die Hauptroutine und damit auch der Task laufen nun lange genug, um dem neuen Thread die Möglichkeit zu geben, das `printf()` auszuführen.

Ein `sleep()` ist aber keine allgemeingültige Lösung, da sich die Laufzeit eines Threads nur selten genau bestimmen lässt. Entsprechend wird eine Möglichkeit benötigt, dem Haupt-Thread mitzuteilen, dass der neu erzeugte Thread beendet wurde. Dafür eignen sich die in den Grundlagen erwähnten Zustandsvariablen. Da das Warten auf das Ende eines Threads eine grundlegende Funktion ist, die in der Praxis häufig benötigt wird, existiert hierfür die nachfolgende Posix-Thread-Funktion.

**`int pthread_join(pthread_t thread, void **value_ptr);`**

Diese Funktion blockiert den aufrufenden Thread so lange, bis der durch den ersten Parameter identifizierte Thread beendet wird. Sollte der Thread bereits beendet sein, so liefert diese Funktion sofort ohne zu warten einen Rückgabewert. Der zweite Parameter, `value_ptr`, kann verwendet werden, um dem wartenden Thread einen Rückgabewert zu übermitteln. Er soll hier nicht weiter betrachtet werden. Mit dieser Funktion kann das vorherige Beispiel wie folgt geschrieben werden:

```
#include <pthread.h>
#include <stdio.h>

void* thread_main(void* arg) {
    printf("%s, hier Thread!\n", (char*)arg);
}

const char* data = "Hallo Welt";

int main() {
    pthread_t th;
    pthread_create(&th, NULL, thread_main, data);
    pthread_join(th, NULL);
}
```

### 12.3.2 Mutex-Funktionen

Um sicherzustellen, dass nur jeweils ein Thread in einen kritischen Bereich eintritt, verwendet Posix Mutexe mit erweiterter Funktionalität. Hier werden die vier essentiellen Funktionen vorgestellt, mit denen einfache Mutexe realisiert werden können, wie sie in Abschnitt 12.2 eingeführt wurden.

**`int pthread_mutex_init(pthread_mutex_t *mutex,  
const pthread_mutexattr_t *attr);`**

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

`pthread_mutex_init()` erzeugt die Datenstruktur, die für den gegenseitigen Ausschluss benötigt wird. `pthread_mutex_destroy()` gibt diese Datenstruktur wieder frei. `pthread_mutex_init()` muss dementsprechend vor der allerersten Verwendung aufgerufen werden. `pthread_mutex_destroy()` darf erst aufgerufen werden, wenn die Mutex nicht mehr verwendet wird. Es bietet sich entsprechend an, die Mutex vor dem Erstellen der Threads in der Hauptroutine zu erzeugen und sie am Ende der Hauptroutine, wenn keine Threads mehr laufen, wieder zu zerstören.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Vor dem Eintritt in den kritischen Bereich wird `pthread_mutex_lock()` aufgerufen. Nach diesem Aufruf ist die Mutex, die durch den Parameter `mutex` definiert wird, für den aufrufenden Thread reserviert. Alle anderen Threads, die nun `pthread_mutex_lock()` mit dem gleichen `mutex` Parameter aufrufen, werden so lange blockiert, bis der Thread die Mutex wieder freigibt.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

`pthread_mutex_unlock()` gibt die im Parameter `mutex` angegebene Mutex wieder für andere Threads frei. Sollten andere Threads zwischenzeitlich `pthread_mutex_lock()` aufgerufen haben und blockiert sein, wird einer von diesen durch den Aufruf von `pthread_mutex_unlock()` wieder freigegeben, sodass er seinen kritischen Abschnitt betreten kann.

### 12.3.3 Funktionen für Zustandsvariablen

```
int pthread_cond_init(pthread_cond_t *cond, const
```

```
pthread_condattr_t *attr);
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

`pthread_cond_init()` erzeugt die Datenstruktur, die die Zustandsvariable im Folgenden identifiziert. `pthread_cond_destroy()` zerstört diese Datenstruktur wieder. Das bei `pthread_mutex_init()` Erwähnte gilt auch hier.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *
mutex);
```

Diese Funktion blockiert den aufrufenden Thread so lange, bis ein anderer Thread `pthread_cond_signal()` oder `pthread_cond_broadcast()` für die mit dem Parameter `cond` identifizierte Zustandsvariable aufruft. Vor dem Aufruf dieser Funktion muss die Mutex, die durch den Parameter `mutex` identifiziert wird, von dem aufrufenden Thread gesperrt werden. `pthread_cond_wait()` gibt die Mutex zwischenzeitlich wieder frei, damit andere Threads ihre kritischen Abschnitte betreten können, während der

Thread wartet. Sobald ein anderer Thread den schlafenden Thread wieder aufweckt, wird die Mutex dem aufgeweckten Thread wieder zugewiesen. Im Standardfall sieht eine Verwendung dieser Funktion wie folgt aus:

```
pthread_mutex_t mutex; // wird irgendwo initialisiert
pthread_cond_t cond;   // -- "" --
char condition=0; // wird in einem anderen Thread auf true gesetzt
                   und danach pthread_cond_signal aufgerufen.
// ...
pthread_mutex_lock(&mutex);
while (!condition) {
    pthread_cond_wait(&cond, &mutex);
}
// hier kann der Thread das machen, was er tun soll
condition = 0;
pthread_mutex_unlock(&mutex);
```

Auf den ersten Blick scheint es unnötig, nach einem Aufruf von `pthread_cond_wait()` nochmals zu überprüfen, ob die Bedingung `condition` erfüllt ist. Es kann aber der Fall eintreten, dass nach dem Aufruf von `pthread_cond_wait()` die Variable `condition` nicht 1 ist. Geschehen kann dies bspw. durch Posix-Signale<sup>12</sup> oder dadurch, dass mehr als nur ein Thread zur gleichen Zeit aufgeweckt wurde.

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Diese beiden Funktionen wecken entweder einen (`pthread_cond_signal()`) oder alle wartenden Threads (`pthread_cond_broadcast()`) auf. Auch diese Funktionen dürfen nur aufgerufen werden, wenn die zu der Zustandsvariablen `cond` zugehörige Mutex im aufrufenden Thread gesperrt wurde. Demnach sieht ein zum obigen Beispiel passender Aufruf wie folgt aus:

```
pthread_mutex_lock(&mutex);
condition = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

### 12.3.4 Beispiel

Im Verzeichnis `threads_basic` im Beispielordner zu diesem Kapitel befindet sich das folgende kleine Testprogramm, das alle hier vorgestellten Funktionen verwendet. Es erzeugt zwei Threads, die beide auf eine gemeinsame Integer-Variable zugreifen.

Ein Thread erhöht die Variable, der andere wartet auf dieses Ereignis und gibt den neuen Wert aus:

<sup>12</sup> Hier nicht weiter erklärte asynchrone Form der Interprozesskommunikation. Die Tastatureingabe von **STRG+C** im Terminal erzeugt z. B. ein **SIGINT**-Signal an den aktuell laufenden Task.

```

#include <pthread.h>
#include <stdio.h>

pthread_mutex_t mutex;
pthread_cond_t cond;
char condition=0;

void *thread1_function(void *val_p) {
    for(;;) {
        pthread_mutex_lock(&mutex);
        while (!condition) {
            pthread_cond_wait(&cond, &mutex);
        }
        printf("Read value %d \n", *((int*)val_p));
        sleep(1);
        condition = 0;
        pthread_mutex_unlock(&mutex);
    }
}

void *thread2_function(void *val_p) {
    for(;;) {
        sleep(1);
        pthread_mutex_lock(&mutex);
        ++(*((int*)val_p));
        printf("Incrementing value, is now %d \n", *((int*)val_p));
        condition = 1;
        pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

int main() {
    pthread_t th1, th2;
    int value = 4;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    printf("Exit Program with Ctrl+C\nInitial value is %d\n", value);

    pthread_create(&th1, NULL, thread1_function, &value);
    pthread_create(&th2, NULL, thread2_function, &value);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mutex);

    return 0;
}

```

Kompiliert wird dieser Quelltext wie auch alle anderen Beispiele dieses Kapitels, indem `make` in dem Verzeichnis aufgerufen wird. Die Ausgabe des Beispielprogramms sieht wie folgt aus:

```

$ ./threads_main
Exit Program with Ctrl+C
Initial value is 4
Incrementing value, is now 5
Read value 5
Incrementing value, is now 6
Read value 6

```



```
Incrementing value, is now 7
Read value 7
Incrementing value, is now 8
Read value 8
Incrementing value, is now 9
Read value 9
^C
```

## 12.4 C++-Schnittstelle

Der vorliegende Abschnitt stellt eine C++-Schnittstelle zu den Posix-Threads, Mutexen und Zustandsvariablen vor, die in den folgenden Kapiteln verwendet wird, um Programme mit mehreren Threads zu entwickeln. Die Schnittstelle beschränkt sich in den meisten Fällen bewusst auf den Standardfall und lässt erweiterte Posix-Thread-Funktionen außer Acht, um einen einfachen Einstieg in die Thread-Programmierung zu ermöglichen.

### 12.4.1 Die Klasse Thread

Die **Thread**-Klasse ist eine abstrakte Klasse.<sup>13</sup> Sie dient nur als Basisklasse für spezifische Thread-Klassen und bietet diesen das Grundgerüst für einen Thread. Sie besitzt die folgende Signatur:

```
virtual void run()= 0;
```

Diese abstrakte virtuelle Methode ist die einzige, die von Unterklassen implementiert werden *muss*, um einen neuen Thread zu erstellen. Sie ist die Hauptroutine des neuen Threads und bestimmt damit dessen Lebensdauer.

```
void start(Prio prio = PRIO_NO);
```

Mit dieser Methode wird ein Thread gestartet. Bei der C++-Schnittstelle sind das Erzeugen eines Threads und das Starten des Threads zwei getrennte Operationen. Bei der C-Schnittstelle der Posix-Threads sind beide zusammengefasst. Der optionale Parameter **prio** gibt die Priorität des Threads an. Er kann einen der folgenden Werte annehmen:

- **PRIO\_NO** (der Standardwert)
- **PRIO\_LOW**
- **PRIO\_NORMAL**
- **PRIO\_HIGH**

<sup>13</sup> Eine Klasse, von der keine Instanzen erzeugt werden können.

Threads mit höherer Priorität werden bei einigen Operationen bevorzugt behandelt. So bekommen sie z. B. bevorzugt den Lock auf eine Mutex. Wenn man hier eine andere Option als den Standard (`PRIO_NO`) angibt, wird der Thread zu einem *Echtzeit-Thread*, der von Linux bevorzugt behandelt wird. Das Erstellen von Echtzeit-Threads benötigt Root-Rechte und das Programm muss zusätzlich zu der `pthread`-Bibliothek noch mit der `rt`-Bibliothek gebunden werden.

```
bool wait(unsigned long time = ULONG_MAX);
```

Diese Methode wird aufgerufen, um auf das Ende des von diesem Objekt repräsentierten Threads zu warten. Nur wenn der optionale Parameter `time` angegeben wird, kann der Rückgabeparameter dieser Funktion `false` sein – dann, wenn der Thread auch nach `time` Mikrosekunden noch läuft.

```
bool isRunning();
```

Der Rückgabewert dieser Methode ist `true`, falls der Thread noch läuft. Diese Methode ist nicht thread-sicher, d. h. das Ergebnis dieser Funktion ist nicht unbedingt korrekt. Sie kann `true` zurückliefern, obwohl der Thread mittlerweile bereits beendet wurde. Dies sollte bei Verwendung dieser Funktion immer berücksichtigt werden.

```
bool isFinished();
```

Gibt `true` zurück, wenn der Thread nicht mehr läuft. Diese Methode ist das Gegenstück zu `isRunning()` und ist ebenfalls nicht thread-sicher.

```
static void setTaskPrio(Prio prio = PRIO_NORMAL);
```

Diese *statische* Methode setzt die Priorität des gesamten *Tasks*. Threads eines hochpriorisierten Tasks werden gegenüber Threads eines niederpriorisierten Tasks vom Prozess-Scheduler bevorzugt behandelt. Da man mit dieser Funktion und einem nicht-kooperativen Thread das System unbenutzbar machen kann, darf in Linux nur der Root-Benutzer die Priorität des Tasks verändern.

Ein minimales Beispiel, das nur einen neuen Thread startet und auf sein Ende wartet, sieht folgendermaßen aus:

```
#include "tools/Thread.h"
#include <iostream>

class HelloWorldThread : public Thread {
public:
    virtual void run() {
        std::cout << "Hallo Welt! " << std::endl;
        sleep(1);
    }
};

int main() {
    HelloWorldThread thread;
    thread.start();
    thread.wait();
    return 0;
}
```

### 12.4.2 Die Klasse Mutex

Eine direkte Umsetzung der Posix-Thread-Mutex-Funktionen in C++ ergibt die folgende Klasse, die im Header `src/tools/Mutex.h` deklariert ist:

```
Mutex();
~Mutex();
```

Im Konstruktor des Objekts wird die Mutex-Variable initialisiert. Im Destruktor wird sie wieder zerstört. Durch die Initialisierung im Konstruktor ist sichergestellt, dass `lock()` und `unlock()` immer auf initialisierten Mutexen operieren.

```
void lock();
void unlock();
```

Nach dem Aufruf der `lock()`-Methode besitzt der aufrufende Thread die Mutex, bis er sie wieder mit `unlock()` freigibt.

Der Kopierkonstruktor `Mutex(const Mutex&)` ist in dieser Klasse nicht von außerhalb aufrufbar, da Mutex-Variablen nur als Referenzen und Pointer übergeben werden sollten. Zu einer Mutex sollte auch nur genau ein `Mutex`-Objekt zugeordnet werden, da sonst die Lebensspanne der Mutex nicht eindeutig bestimmt werden kann.

Die `Mutex`-Klasse ist auf den ersten Blick einfach zu verwenden: Vor dem Eintritt in den kritischen Abschnitt wird die `lock()`-Methode aufgerufen, nach dem Abschnitt die `unlock()`-Methode. Problematisch ist, dass es viele Wege aus dem kritischen Abschnitt gibt, von denen die meisten nicht offensichtlich sind. Das folgendes Beispielfragment, in dem der Zugriff auf eine gemeinsam genutzte Variable durch eine Mutex abgesichert wird, verdeutlicht dieses Problem:

```
Mutex m;
char* shared_data = NULL;
extern char* create_shared_data() throw(std::bad_alloc);
...
void function critical_section(){
    m.lock(); // Mutex bei Funktionseintritt holen
    if (!shared_data) {
        shared_data = create_shared_data(); // (b)
    }
    ...
    if (error) {
        return; // (a)
    }
    ...
    m.unlock(); // Beim Verlassen Mutex freigeben
}
```

Intention des Programmierers ist es, beim Eintritt in die Funktion `critical_section()` die Mutex zu bekommen und sie beim Verlassen wieder freizugeben. Im Beispielcode wurde dies aber im Fehlerfall bei (a) vergessen.

Der Fehler bei (b) ist subtiler: Die Funktion `create_shared_data()` kann eine `std::bad_alloc`-Ausnahme werfen, wenn nicht mehr genügend Speicher verfügbar ist. Auch in diesem Fall wird die Funktion `critical_section()` ohne Freigabe der Mutex verlassen.

In diesem Beispiel kann man erkennen, dass es einige Möglichkeiten gibt, die `Mutex`-Klasse falsch zu verwenden. Speziell die in C++ hinzugekommenen Ausnahmen machen es nicht immer leicht, jeden Weg aus der Funktion abzusichern. Man kann allerdings dem C++-Compiler die Verantwortung übertragen. Die Mutex lässt sich automatisch entsperren, indem man die Tatsache nutzt, dass der Destruktor eines in einer Funktion lokal definierten Objekts immer aufgerufen wird, sobald der Programmfluss den Gültigkeitsbereich dieser Funktion verlässt.<sup>14</sup> Wie und wo das geschieht, ob per `return`-Befehl oder durch eine Ausnahme, ist nicht relevant.

Unter Nutzung dieses Sachverhaltes erhält die Schnittstelle die Klasse `MutexLocker` mit der folgenden Signatur:

```
MutexLocker(Mutex& m);
```

Der Konstruktor der Klasse bekommt die `Mutex` übergeben, auf der dieses Objekt operieren soll. Er ruft auch sofort die `lock()`-Methode von dieser `Mutex` auf.

```
~MutexLocker();
```

Der Destruktor gibt die `Mutex` wieder frei.

```
void unlock();
```

Diese Methode ruft die gleichnamige Methode der `Mutex` auf und gibt somit die `Mutex` vorzeitig wieder frei. Nach einem Aufruf dieser Funktion gibt der Destruktor die `Mutex` natürlich kein zweites Mal frei.

Die richtige Verwendung dieser Klasse kann den Einsatz von Mutexen wesentlich robuster machen. So ist das folgende Beispiel eine robuste und korrekte Version des obigen Beispiels:

```
void function critical_section(){
    MutexLocker lock(m); // Mutex bei Funktionseintritt holen
    if (!shared_data) {
        shared_data = create_shared_data();
    }
    ...
    if (error) {
        return;
    }
    ...
    // Beim Verlassen des Gültigkeitsbereichs Mutex freigeben
}
```

Was allerdings vermieden werden sollte, ist das Zwischenspeichern einer solchen `MutexLocker`-Variable oder eines Pointers auf einen `MutexLocker`. Die

<sup>14</sup> Dies ist nicht nur auf Funktionen beschränkt, sondern gilt allgemein für jeden Gültigkeitsbereich (engl. *Scope*).

Klasse ist nur dafür gedacht, lokal in einer Funktion (oder in einem Teil einer Funktion) verwendet zu werden. Aus diesem Grund ist sie selbst auch nicht thread-sicher.

### 12.4.3 Die Klasse `WaitCondition`

Die Klasse `WaitCondition` implementiert die Posix-Zustandsvariablen als C++-Klasse. Sie wird in der Header-Datei `src/tools/WaitCondition.h` deklariert und bietet dem Anwender die folgende API:

```
WaitCondition(Mutex& m);
~WaitCondition();
```

Der Konstruktor initialisiert die Datenstruktur, während sie im Destruktor zerstört wird. Dem Konstruktor muss eine `Mutex`-Instanz übergeben werden, die mit dieser Zustandsvariablen zusammen verwendet wird.

```
void wait();
```

Der aufrufende Thread wird so lange blockiert, bis ein anderer Thread `signalOne()` oder `signalAll()` aufruft. Vor dem Aufruf dieser Methode *muss* die im Konstruktor angegebene `Mutex` gesperrt werden. Diese Methode reiht den Thread in die Warteschlange ein, ruft `unlock()` auf die `Mutex` auf und blockiert den aufrufenden Thread. Wenn der Thread aus der Methode `wait()` zurückkehrt, dann ist sichergestellt, dass er die `Mutex` wieder erhalten hat. Nach dem Aufruf dieser Methode muss die `Mutex`-Variable folglich freigegeben werden, trotz eines zwischenzeitlichen Freigebens und Wiedererhaltens der `Mutex`. Eine typische Verwendung dieser Funktion sieht wie folgt aus:

```
Mutex m;
WaitCondition w(m);
...
m.lock();
w.wait();
m.unlock();
```

```
bool wait(const timespec * abstime);
```

Diese Methode hat die gleiche Funktionalität wie die Methode `wait()`, allerdings wartet der Thread nicht unendlich lange, sondern maximal bis zum absoluten Zeitpunkt `abstime`. Die Datenstruktur `timespec` wird in der Posix-Header-Datei `time.h` deklariert und sieht aus wie folgt:

```
typedef struct {
    long tv_sec;
    long tv_nsec;
}
```

Die Header-Datei `timing_functions.h`<sup>15</sup> enthält die folgenden Funktionen, die den Umgang mit dieser Datenstruktur erleichtern:

<sup>15</sup> Zu finden unter `<embedded-linux-dir>/src/tools/`.

```
timespec& getCurrentTime(timespec& sp);
```

Diese Funktion füllt `sp` mit der aktuellen Zeit. Das so modifizierte `sp` wird von dieser Funktion zurückgegeben, sodass es sofort weiterverwendet werden kann.

```
void normalizeTimespec(timespec& sp);
```

Durch die Aufteilung in Sekunden und Nanosekunden kann ein und derselbe Zeitpunkt durch unterschiedliche Kombinationen von Sekunden und Nanosekunden dargestellt werden. Z.B. sind die beiden Zeitpunkte `tv_sec = 42, tv_nsec = 0` und `tv_sec = 41, tv_nsec = 1 000 000 000` nicht identisch, bilden aber den gleichen Zeitpunkt ab. `normalizeTimespec` stellt sicher, dass `tv_nsec` immer kleiner als 1 000 000 000 ist. Ein `timespec` muss immer normalisiert sein, wenn man ihn als Zeitpunkt in einer Funktion wie der hier beschriebenen `wait(const timespec* abstime)` verwendet.

Das folgende Beispiel zeigt, wie man ein `timespec` für 0,5 Sekunden in der Zukunft erstellt:

```
timespec ts;
getCurrentTime(ts);
ts.tv_nsec += 500000000;
normalizeTimespec(ts);
```

```
long long getNanoseconds(const timespec& sp);
```

Gibt den Wert dieses `timespecs` in Nanosekunden zurück. Der Rückgabewert entspricht demzufolge  $(sp.tv\_sec \cdot 1\,000\,000\,000 + sp.tv\_nsec)$ .

```
void signalOne();
```

Es wird genau ein wartender Thread aufgeweckt. Die `Mutex` muss vor dem Aufruf dieser Methode von dem aufrufenden Thread reserviert worden sein.

```
void signalAll();
```

Es werden alle wartenden Threads aufgeweckt. Auch diese Methode erfordert, dass die `Mutex` dem aufrufenden Thread zugewiesen wurde.

Als exemplarische Verwendung der Klasse `WaitCondition` findet man bei den Beispielquelltexten unter `queue/Queue.h` die Implementierung einer synchronisierten Warteschlange. Mehrere Produzenten könnten die Warteschlange füllen, während gleichzeitig mehrere Konsumenten von ihr Produkte erhalten können. Die Klasse umschifft alle möglichen Race Conditions automatisch und bietet dem Anwender die folgende, einfache Schnittstelle:

```
template<class item_t> Queue(unsigned int max_size = 0);
```

Dem Konstruktor der Klasse kann optional durch den Parameter `max_size` eine maximale Größe der Warteschlange mitgegeben werden. Ist diese Größe erreicht, so werden die Produzenten so lange blockiert, bis ein Konsument wieder einen Platz in der Warteschlange freigegeben hat.

```
void put(const item_t& item);
```

Sobald ein neues Produkt erstellt wurde, kann der Produzent es mit dieser Methode in die Warteschlange einreihen. Sollte bereits ein Konsument auf ein neues Element in der Warteschlange warten, so wird er hierdurch aufgeweckt.

```
item_t get();
```

```
void item_finished();
```

Die Methode `get` entfernt das erste Element aus der Warteschlange und liefert es dem aufrufenden Konsumenten zurück. Sollte die Warteschlange leer sein, so wird der Konsument so lange blockiert, bis die Warteschlange wieder ein Element enthält.

Nachdem ein Konsument mit `get` ein Element zur Bearbeitung erhalten hat, sollte er `item_finished` aufrufen, sobald er mit der Bearbeitung des Elements fertig ist.

```
void wait();
```

Der aufrufende Thread wird so lange blockiert, bis die Warteschlange leer ist und für jeden `get`-Aufruf ein entsprechender `item_finished`-Aufruf erfolgt ist. Sollte dies beim Aufruf bereits der Fall sein, wird der Thread nicht blockiert. Wenn sichergestellt ist, dass die Produzenten ihre Arbeit vollbracht haben, kann mit dieser Methode auf das Ereignis gewartet werden, dass die Konsumenten die Warteschlange abgearbeitet haben. Wenn nicht sichergestellt ist, dass keine neuen Elemente in die Warteschlange gelangen, wartet diese Methode nicht unbedingt den richtigen Zeitpunkt ab.

Im Verzeichnis `queue` findet sich ein Beispielprogramm, das die korrekte Verwendung der Warteschlange demonstriert. Der Rest dieses Abschnitts widmet sich aber der Synchronisationslogik, die in der `Queue`-Klasse gekapselt ist und diese thread-sicher macht.

Alle oben beschriebenen Methoden der Warteschlange müssen selbstverständlich durch gegenseitigen Ausschluss geschützt werden, da sie alle interne Datenstrukturen der Warteschlange verändern. Dafür besitzt die Warteschlange eine `Mutex`-Variable, die in den Methoden jeweils durch eine lokale `MutexLocker`-Variable für den aufrufenden Thread reserviert wird. Als Beispiel sei hier die `wait()`-Methode gezeigt:

```
void Queue::wait()
{
    MutexLocker ml(mutex);
    // wait until all consumers reported that their processing is done
    while (unfinished_items > 0) {
        cond_all_items_finished.wait(); // automatically unlocks the mutex
        while waiting
    }
}
```

Diese Methode zeigt die erste Verwendung einer Zustandsvariablen, um auf ein Ereignis zu warten: Solange die Zählvariable `unfinished_items` größer als Null ist, wird gewartet, bis sie Null erreicht. Auch hier gilt (genau wie bei den Posix-Zustandsvariablen), dass nach dem `wait`-Aufruf auf die Zustandsvariable nochmals geprüft werden muss, ob die Bedingung (`unfinished_items <= 0`) wirklich noch zutrifft. Zwischen dem Signal, dass die Zählvariable Null erreicht hat und dem Aufwecken des darauf wartenden Threads, könnte die Warteschlange wieder mit neuen Elementen gefüllt worden sein.

Die Warteschlange besitzt noch zwei weitere `WaitCondition`-Variablen, die in der `put()`- und `get()`-Methode die Ereignisse *Die Warteschlange ist nicht mehr leer* (`cond_not_empty`) und *Die Warteschlange ist nicht mehr voll* (`cond_not_full`) signalisieren.

#### 12.4.4 Die Klasse `PeriodicThread`

Viele Aufgaben eines eingebetteten Systems lassen sich als periodische Prozesse darstellen. Das Auslesen eines Sensors, die Ansteuerung eines Servomotors (siehe Abschnitt 12.5) oder auch Hintergrundberechnungen sind nur einige Beispiele. Für diese Zwecke wurde die Klasse `PeriodicThread`, eine Unterklasse von `Thread`, in die C++-Schnittstelle aufgenommen. Die folgende Beschreibung der Klasse beschränkt sich auf die neu hinzugekommenen Methoden. Als Unterklasse von `Thread` besitzt sie weiterhin auch alle Methoden ihrer Oberklasse. Hier sollte noch angemerkt werden, dass `PeriodicThread` die `run()`-Methode bereits implementiert; Unterklassen von `PeriodicThread` sollten diese Methode nicht wie andere `Thread`-Unterklassen überschreiben.

```
PeriodicThread(long long period, wait_type_t wait_type =
    WAIT_ABS_NANOSLEEP);
```

Der Konstruktor der Klasse bekommt die Periode, mit der dieser Thread ausgeführt werden soll, in Nanosekunden übergeben. Die Frequenz der Ausführung kann nur hier einmalig im Konstruktor spezifiziert werden, da eine dynamische Anpassung der Ausführungsfrequenz während der Laufzeit des Threads in den meisten Fällen nicht notwendig ist. Der zweite Parameter des Konstruktors gibt an, mit welchem Mechanismus die Periode eingehalten werden soll. Die drei zur Verfügung stehenden Alternativen werden bei der Methode `wait_for_next_turn()` genauer erklärt.

```
void pause();
void resume();
void terminate();
```

Mithilfe der Methode `pause()` wird der periodische Thread pausiert, bis er entweder beendet oder mit der `resume()`-Methode wieder fortgesetzt wird. Auch während der Thread pausiert, wird die Periode weiter eingehalten,



sodass `resume()` den Thread zu einem Zeitpunkt wieder fortsetzen lässt, der mit der ursprünglichen Periode übereinstimmt.

Gestartet wird ein `PeriodicThread` genauso wie alle anderen `Thread`-Klassen mit seiner `start()`-Methode.

Die Methode `terminate()` beendet den periodischen Thread, wobei sichergestellt ist, dass dies nicht während der Ausführung einer Periode geschieht.

```
virtual void do_turn()= 0;
```

Diese Methode wird zum Anfang jeder Periode aufgerufen. Sie muss von Unterklassen überschrieben werden, um die gewünschte periodische Aktivität auszuführen.

```
void wait_for_next_turn();
```

Diese Methode enthält die Logik, die für das Warten auf die nächste Periode notwendig ist. Es sind die drei folgenden unterschiedlichen Arten definiert: `WAIT_NANOSLEEP`, `WAIT_NANOSLEEP_AND_BUSYWAIT` und `WAIT_ABS_NANOSLEEP`.

Alle drei Arten verwenden die Standardfunktion `clock_nanosleep()`, die ein Teil der Posix-Echtzeiterweiterung ist.<sup>16</sup> `WAIT_NANOSLEEP` und `WAIT_NANOSLEEP_AND_BUSYWAIT` verwenden die Funktion mit einer relativen Wartezeit, `WAIT_ABS_NANOSLEEP` verwendet die Funktion mit einem absoluten Zeitpunkt. Um die relative Wartezeit zu bestimmen, muss ein zusätzlicher Aufruf von `clock_gettime()` direkt vor dem Aufruf von `clock_nanosleep()` erfolgen, um die Wartezeit bis zur nächsten Periode zu errechnen. Zwischen den beiden Betriebssystemaufrufen kann der Prozess-Scheduler den Thread unterbrechen und zu einem späteren Zeitpunkt wieder fortführen. Damit stimmt dann die berechnete Wartezeit nicht mehr und die Periode wird nicht eingehalten. Das Warten auf einen absoluten Zeitpunkt umgeht diese Problematik, da hier nur einmal zu Beginn des Threads die aktuelle Zeit durch einen Betriebssystemaufruf ermittelt werden muss. Danach kann die nächste Periode durch Addition der Periodendauer zu dieser Uhrzeit errechnet werden.

Wenn die Auflösung der Uhr nicht viel größer als die gewünschte Periodendauer ist, kann es zu systematischen Verzögerungen kommen. Möchte man etwa eine Periode von 15 ms erhalten, das verwendete System besitzt aber nur eine Timer-Auflösung von 10 ms, so kann `clock_nanosleep()` nur eine Periodendauer von 20 ms erreichen. Die 15 ms-Periode kann in diesem Fall mit einem Trick doch noch erreicht werden: Man lässt `clock_nanosleep()` nicht bis zur nächsten Periode warten, sondern nur gerade so lange, dass der Thread kurz vor der Periode wieder aufwacht. Die restliche Zeit bis zum Beginn der nächsten Periode wird durch konstantes Abfragen der Uhrzeit und Vergleich mit dem Start der nächsten Periode verbracht. Im

<sup>16</sup> [http://www.opengroup.org/onlinepubs/009695399/functions/clock\\_nanosleep.html](http://www.opengroup.org/onlinepubs/009695399/functions/clock_nanosleep.html).

Beispiel würde der Thread entsprechend die letzten 5 ms durch dieses sog. *busy waiting* abmessen.

Das folgende Programm `threads_periodic` zeigt die Verwendung dieser Klasse. Ein umfangreicheres Programm mit mehreren `PeriodicThreads` wird im nachfolgenden Abschnitt vorgestellt.

```
#include "tools/PeriodicThread.h"
#include <iostream>

using namespace std;

struct MyThread: public PeriodicThread {
    MyThread() : PeriodicThread(500000000LL) {} // 0.5s
    virtual void do_turn() {
        cout << "MyThread's turn" << endl;
    }
};

int main() {
    MyThread t;
    cout << "PeriodicThread example\n\n" << endl;
    t.start();
    char c=0;
    while (c!='q'&&c!='Q') {
        cout << "Q = Quit\nP = Pause\nR = Resume\n" << endl;
        cin >> c;
        switch (c) {
            case 'p': case 'P':
                t.pause(); break;
            case 'r': case 'R':
                t.resume(); break;
        }
    }
    t.terminate();
    t.wait();
    return 0;
}
```

## 12.5 Anwendungsbeispiel: Servo-Ansteuerung

Zum Abschluss soll ein praktisches Beispiel die Anwendung der in diesem Kapitel vorgestellten Konzepte und Schnittstellen verdeutlichen: Das Beispielprogramm steuert einen Modellbauservo an und nimmt Steuerkommandos des Anwenders an der Konsole entgegen. Optional kann es die aktuelle Position des Servos auf einem I<sup>2</sup>C-Display anzeigen und ist weiterhin auch in der Lage, die Genauigkeit der Servo-Ansteuerung zu messen. Das Beispiel befindet sich im Unterverzeichnis `servo` bei den Beispielen dieses Kapitels. Es benötigt Root-Rechte, da es erstens direkt auf die Hardware zugreift und zweitens seinen Threads Echtzeitpriorität zuweist. Mit der Komandozeilenoption `-r` teilt es außerdem dem kompletten Task die höchste Priorität zu.

Im Programm stehen die folgenden Kommandos zur Benutzerinteraktion zur Verfügung:

```

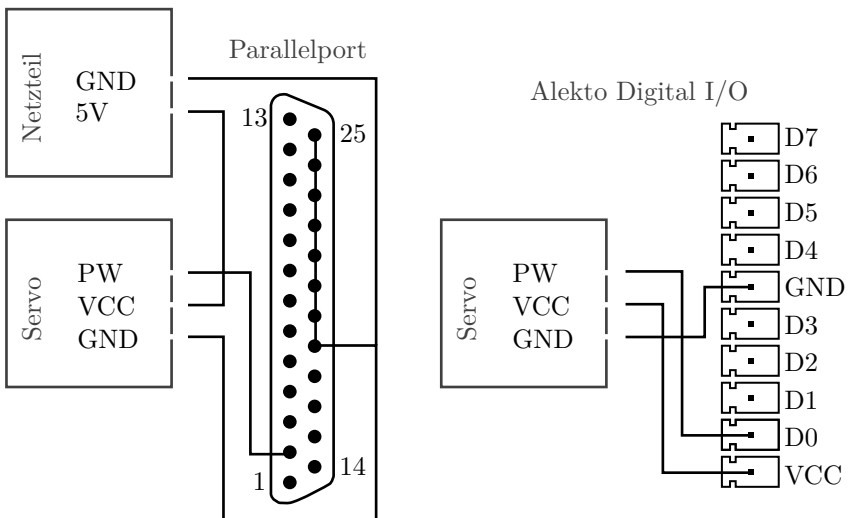
$ cd $EMBEDDED_LINUX/examples/threads/servo/
$ make
$ sudo ./servo_tester -r
Task priority set to REALTIME
CLOCK_REALTIME resolution = 1ns

Q      - Quit application
P      - Pause Servo Thread
R      - Resume Servo Thread
T      - Start a timing (written to timing.data)
0-9    - Set the servo position (0=leftmost, 9=rightmost)
A      - Automatic servo movement

```

Im nachfolgenden Abschnitt werden Entwurf und Quellcode des Beispiels genauer betrachtet. Zuvor werden die Kenngrößen des Servomotors und die Art der Ansteuerung vorgestellt.

### 12.5.1 Servo-Anbindung an einen PC

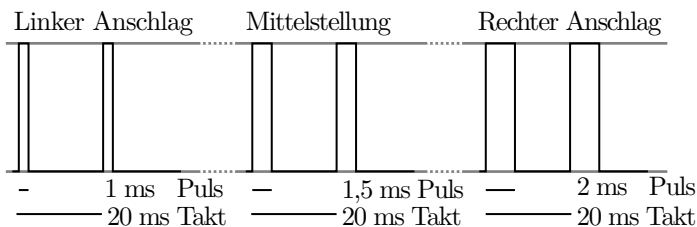


**Abb. 12.4.** Anbindung eines Modellbau-Servos an den Parallelport bzw. an den digitalen I/O-Port des Alekto.

Ein Servo ist ein aus dem Modellbau bekannter Stellmotor, der durch eine Pulsbreitenmodulation angesteuert wird. Typische Taktraten betragen 50 Hz (entspricht einem Puls alle 20 ms) mit einer wechselnden Pulsbreite von ca. 1 ms bis ca. 2 ms.<sup>17</sup> Er findet im Modellbau Verwendung als einfach anzusteuender

<sup>17</sup> Die genauen Werte sind vom verwendeten Modell abhängig.

Stellmotor bspw. für die Lenkung eines ferngesteuerten Autos. Die meisten Servos arbeiten mit einer Spannung und einem Taktsignal von 4,8 V bis 6 V, wodurch sie ohne Pegelkonvertierung mit den bei Computeranschlüssen üblichen High-Pegeln von 5 V angesteuert werden können. Abbildung 12.4 zeigt, wie ein Servo an den Parallelport oder den digitalen I/O-Port des Alekto (siehe Kapitel 5) angeschlossen werden kann. Die Steuerleitung kann in beiden Fällen mit jeder der acht Datenleitungen verbunden werden. In der Abbildung ist sie jeweils mit der ersten Datenleitung verbunden. Versorgt wird der Servo über eine externe 5 V-Stromquelle. Hierfür kann z. B. ein Festplattenstromanschluss des PC-Netzteils dienen. Die übliche Farbkodierung der Anschlussleitungen eines Servos ist wie folgt: Masse (GND) – schwarz, Versorgungsspannung (VCC) – rot, Steuerleitung (PW) – braun/gelb.



**Abb. 12.5.** Das pulsbreitenmodulierte Steuersignal für einen Servo (die Pulsbreite ist zur Veranschaulichung stark vergrößert dargestellt).

Die Generierung des Steuersignales für einen Servo mithilfe eines PCs ist ebenso einfach wie die Anbindung des Servos an den PC: Alle 20 ms wird für 1 ms bis 2 ms (je nachdem, welche Stellung gewünscht wird) das Steuersignal auf 5 V gesetzt, den Rest der Zeit verbleibt es auf 0 V. Abbildung 12.5 zeigt das Signal für den Fall der maximalen Auslenkung nach links (Pulsbreite 1 ms), der Mittelstellung (Pulsbreite 1,5 ms) und der maximalen Auslenkung nach rechts (Pulsbreite 2 ms).

### 12.5.2 Software-Entwurf zum Beispiel

Der grundlegende Entwurf steht schnell fest: Der Servo wird mithilfe eines periodischen Threads angesteuert. Der Thread hat eine feste Periode von 20 ms und setzt für eine variable Zeitspanne die Steuerleitung des Servos auf High-Pegel. Der Quelltext hierzu sieht aus wie folgt:

```
void ServoThread::do_turn() {
    io_set_high(); // begin of peak
    timespec start_period, timeout, now;
    getCurrentTime(start_period);
    timeout = start_period;
    timeout.tv_nsec += get_peak_in_nanoseconds();
```

```

    normalizeTimespec(timeout);
    while (getCurrentTime(now) < timeout)
    ;
    io_set_low(); // end of peak
}

```

Die Funktionen `io_set_high()` und `io_set_low()` setzen die Steuerleitung auf High-Pegel respektive Low-Pegel.<sup>18</sup> Die Funktion `get_peak_in_nanoseconds()` gibt die aktuelle Pulsbreite in Nanosekunden zurück (entsprechend Werte zwischen 1 000 000 und 2 000 000).

Der aufmerksame Leser wird erkennen, dass die Pulsbreite nicht mittels der vom Betriebssystem zur Verfügung gestellten Wartemechanismen (siehe Abschnitt 12.4.4) gemessen, sondern hier auf aktives Warten zurückgegriffen wird. Diese Technik wird verwendet, da Linux auf einigen Plattformen<sup>19</sup> nur eine zeitliche Auflösung von 10 ms besitzt. Ein `nanosleep()` oder `usleep()`, das eine kleinere Zeitspanne warten soll, wird auf solchen Systemen entsprechend transparent auf 10 ms vergrößert. Entsprechend würde das generierte Signal auf diesen Systemen 50% der Zeit auf dem High-Pegel verweilen. Mit einer solchen Ansteuerung kommt der Servo nicht zurecht und verweigert seinen Dienst. Um auch auf solchen Plattformen ein valides Ansteuersignal generieren zu können, wird hier auf aktives Warten zurückgegriffen.

Der Quelltext für die Überwachung der Periodendauer des Servo-Threads wird nach dem Senden des Pulses hinzugefügt. Dieser Bereich ist nicht zeitkritisch (für ihn stehen immerhin bis zu 19 ms zur Verfügung), sodass erst hier die Mutex reserviert wird, um die von zwei Threads gemeinsam genutzte Datenstruktur zur Messung der Periodendauer abzusichern.

```

void ServoThread::do_turn() {
    timespec start_period;
    getCurrentTime(start_period);

    // ... send pulse

    {
        MutexLocker m(m_stats_mutex);
        long long start = getNanoseconds(start_period);
        if (m_measurements_index > -1) {
            m_measurements[m_measurements_index] = start - m_last_period_start;
            ++m_measurements_index;
            if (m_measurements_index == NUM_MEASUREMENTS) {
                m_measurements_index = -1;
                m_stats_finished.signalAll();
            }
        }
        m_last_period_start = start;
    }
}

```

<sup>18</sup> Definiert werden beide im Beispiel in der Datei `io.cpp`.

<sup>19</sup> Eine dieser Plattformen ist der von diesem Programm unterstützte Alekto (siehe Kapitel 5).

Zuerst wird mit einem `MutexLocker` die `Mutex` gesperrt, die den Zugriff auf die Messstruktur `m_measurements` absichert. Wenn der Anwender eine Zeitmessung angefordert hat (`m_measurements_index`  $\neq -1$ ), so wird der Messstruktur die Differenz zwischen dem Beginn der aktuellen Periode und dem Beginn der vorherigen Periode hinzugefügt. Falls die Messung beendet ist, wird der auf das Ende der Messung wartende Thread aufgeweckt.

Eine Messung wird demnach gestartet, indem die Zählvariable `m_measurements_index` auf 0 gesetzt wird. Danach wartet der aufrufende Thread auf das Signal, welches ein Ende der Messung anzeigt, um die Daten danach auszulesen.

```
ServoThread::measurment_t* ServoThread::get_stats() {
    measurment_t* measurements;
    m_stats_mutex.lock();
    m_stats_finished.wait();
    measurements = m_measurements;
    if (m_measurements == &m_measurements1[0]) {
        m_measurements = &m_measurements2[0];
    } else {
        m_measurements = &m_measurements1[0];
    }
    m_measurements_index = -1;
    m_stats_mutex.unlock();

    measurment_t* measurements_copy = new measurment_t[NUM_MEASUREMENTS];
    memcpy(measurements_copy, measurements, NUM_MEASUREMENTS * sizeof(
        measurment_t));

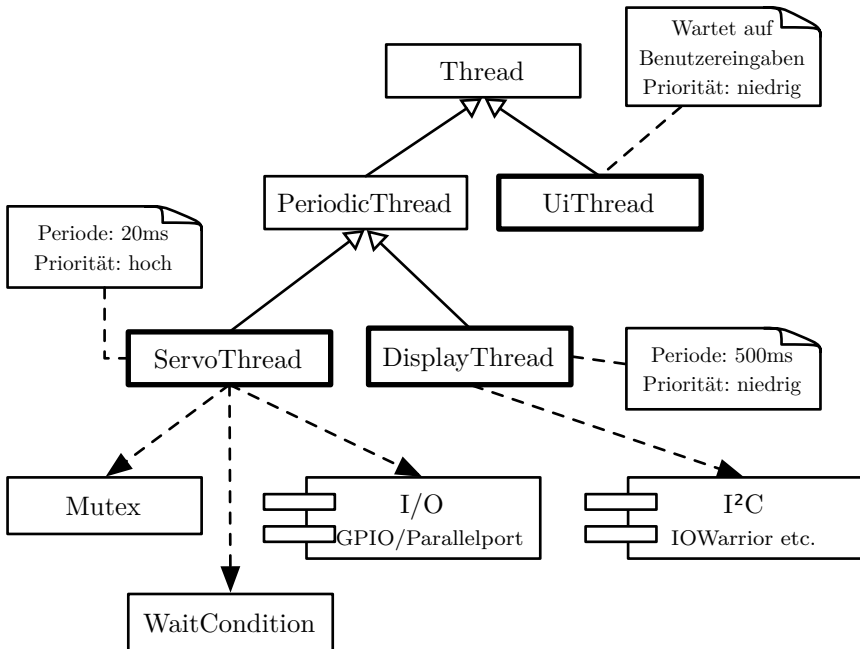
    return measurements_copy;
}
```

Um die `Mutex` möglichst zeitnah wieder freizugeben, wird nicht mit einer, sondern mit zwei Messstrukturen gearbeitet, die hier einfach vertauscht werden. Dadurch kann die `Mutex` bereits freigegeben werden, bevor eine Kopie der Messdaten angelegt wird.

Den komplette Quelltext inklusive den hier nicht näher beschriebenen Klassen `UiThread` (die die Benutzerinteraktion koordiniert) und `DisplayThread` (für die Ausgabe auf dem I<sup>2</sup>C-Display) sowie einige Hilfsfunktionen findet man wie eingangs erwähnt im Beispiel-Verzeichnis `servo`. Abbildung 12.6 erleichtert das Verständnis der Implementierung durch einen Überblick in UML-Notation über die Klassen dieses Beispiels und deren Beziehungen zu anderen Komponenten.

### 12.5.3 Linux und Echtzeitfähigkeit

Im nachfolgenden Abschnitt werden die Ergebnisse der Zeitmessung auf einem Testsystem erläutert. Zuvor soll ein kurzer Überblick über aktuelle Projekte gegeben werden, die Linux um Echtzeitfähigkeit erweitern.



**Abb. 12.6.** Entwurf des Beispiels: Dargestellt sind die für das Beispiel entwickelten Thread-Klassen und ihre Verbindungen zu den in diesem Kapitel vorgestellten Klassen. Die beiden Komponenten I/O und I<sup>2</sup>C werden in anderen Kapiteln dieses Buches erklärt (siehe Abschnitt 9.3 und Abschnitt 5.5.2).

Linux ist als konventionelles Betriebssystem darauf optimiert, einen möglichst hohen Prozess-Durchsatz zu erreichen. Deshalb konnte der Prozess-Scheduler traditionell nur in den folgenden Fällen einen Thread unterbrechen:

- Wenn der Thread im User-Space läuft.
- Wenn der Thread wieder von einem Betriebssystemaufruf in den User-Space zurückkehren will.
- Wenn ein Thread in ausgewählte Bereiche des Kernel-Space eintritt und z. B. auf eine Mutex wartet.
- Wenn der Thread freiwillig die Kontrolle an den Prozess-Scheduler übergibt.

Seit einiger Zeit aber ist der 2.6er-Linux-Kernel mithilfe der Option `CONFIG_PREEMPT` standardmäßig so eingestellt, dass ein Thread auch in vielen Bereichen des Kernels vom Prozess-Scheduler unterbrochen werden kann. Diese Option beeinflusst unter Umständen den Gesamtdurchsatz des Systems negativ, verbessert aber die durchschnittliche Latenz auf den einstelligen Mil-

lisekundenbereich.<sup>20</sup> Interruptbehandlungen und kritische Bereiche der Kernelmodule können nicht unterbrochen werden und entsprechend können maximale Latenzen von mehreren hundert Millisekunden auftreten.

Tabelle 12.1 stellt einige Erweiterungen und Modifikationen vor, die diese maximale Latenz verringern können und so die Entwicklung von echtzeitfähigen Anwendungen unter Linux ermöglichen. Hierbei lassen sich zwei Strategien festhalten: Der `CONFIG_PREEMPT_RT` Patch verfolgt den Ansatz des Standard-Linux-Kernels konsequent weiter, sodass der Kernel in noch mehr Bereichen unterbrechbar ist. Dies verringert die maximale Latenz. Hauptsächlich wegen unvorhersehbaren Interrupts kann aber keine Latenz garantiert werden.

Die anderen Projekte gehen einen Schritt weiter und verlagern die Kontrolle über die Hardware (und hier vor allem über die Interrupts) zu einer Schicht unterhalb des Linux-Kernels. Der Linux-Kernel ist bei ihnen nur ein spezieller RT<sup>21</sup>-Task, der nur dann die Kontrolle erhält, wenn kein anderer RT-Task bedient werden muss.

#### 12.5.4 Zeitmessung

Als Benchmarksystem für die Zeitmessung soll hier das in Abschnitt 2.6 vorgestellte Dual-Atom-Mainboard D945GCLF2 von Intel dienen. Seine zwei 1,6 GHz schnellen Kerne mit Hyper-Threading<sup>22</sup> und die x86-Architektur mit High-Resolution-Timer (Timerauflösung: 1 ns) versprechen geringe Latenzzeiten. Ein Ubuntu 8.04 (LTS) auf einer 80 GB 2,5 Zoll Festplatte dient als Testumgebung.

Canonical Ltd.<sup>23</sup> stellt für diese Ubuntu-Version einen alternativen Linux-Kernel zur Verfügung, der den `CONFIG_REEMPT_RT` Patch bereits enthält. Er lässt sich mit dem folgenden Befehl auf der Konsole installieren und ist danach im Bootloader GRUB beim Starten des Systems auswählbar.

```
$ sudo apt-get install linux-rt
```

Untersucht werden im Folgenden die vier Konfigurationen, die sich ergeben, wenn das Programm jeweils einmal mit und ohne Echtzeitpriorität auf dem Standard-Ubuntu-Kernel (2.6.24-generic) und dem Kernel mit aktiviertem `CONFIG_PREEMPT_RT` Patch (2.6.24-rt) ausgeführt wird.

<sup>20</sup> Voraussetzung hierfür ist, dass Linux auf dem Zielsystem einen High-Resolution-Timer unterstützt. Ohne High-Resolution-Timer beträgt die Standardauflösung 10 ms.

<sup>21</sup> RT steht hier und im Folgenden für engl. *Real-time: Echtzeit*.

<sup>22</sup> Virtualisierungstechnik von Intel, die einen Kern virtuell verdoppelt, sodass zwei Threads quasi gleichzeitig auf dem Kern ausgeführt werden können.

<sup>23</sup> Das Unternehmen, das Ubuntu frei zur Verfügung stellt.



**RTAI** (Real-Time Application Interface): <http://www.rtai.org/>

- Unterstützt die Programmentwicklung gem. harten Echtzeitanforderungen.
- Unterstützte Plattformen: x86, PowerPC.
- Mit der Erweiterung LXRT können RT-Tasks im User-Space ausgeführt werden. Ohne diese Erweiterung müssen RT-Programme als Kernelmodule implementiert werden.
- Enthält viele echtzeitfähige Treiber wie RT-Socket-CAN, RTnet, RT-FireWire.
- Mit dem Entwicklungstool RTAI-Lab lassen sich direkt aus Blockdiagrammen RT-Programme kompilieren.

**Xenomai**: <http://www.xenomai.org/>

- Unterstützt die Programmentwicklung gem. harten Echtzeitanforderungen.
- Unterstützte Plattformen: x86, PowerPC, ARM, Blackfin.
- APIs von verschiedenen anderen RT-Betriebssystemen werden durch sog. *Skins* unterstützt. Eine Migration bestehender RT-Anwendungen von anderen RT-Betriebssystemen ist deswegen einfach.
- Enthält viele echtzeitfähige Treiber wie RT-Socket-CAN, RTnet, RT-FireWire.
- RT-Threads werden automatisch zu normalen Linux-Threads, wenn sie Linux Systemaufrufe tätigen.
- Von RTAI abgespaltet, entsprechend existieren viele Gemeinsamkeiten.

**RTLinux/GPL**: <http://www.rtlinux-gpl.org/>

- Unterstützt die Programmentwicklung gem. harten Echtzeitanforderungen.
- Unterstützte Plattformen: x86.
- Basiert auf einem Patent der Firma FSMLabs, das allerdings für Weiterentwicklungen unter GPL-Lizenz ohne Lizenzkosten verwendet werden darf.

**CONFIG\_PREEMPT\_RT Patch**: <http://rt.wiki.kernel.org/>

- Unterstützt die Programmentwicklung gem. weichen Echtzeitanforderungen.
- Unterstützte Plattformen: x86, ARM.
- Direkteste Integration in den Linux-Kernel. Viele Modifikationen dieses Patches sind bereits in den Standard-Kernel aufgenommen worden.

**Tabelle 12.1.** Freie Echtzeiterweiterungen für Linux: Eine Übersicht der verbreitetsten Alternativen.

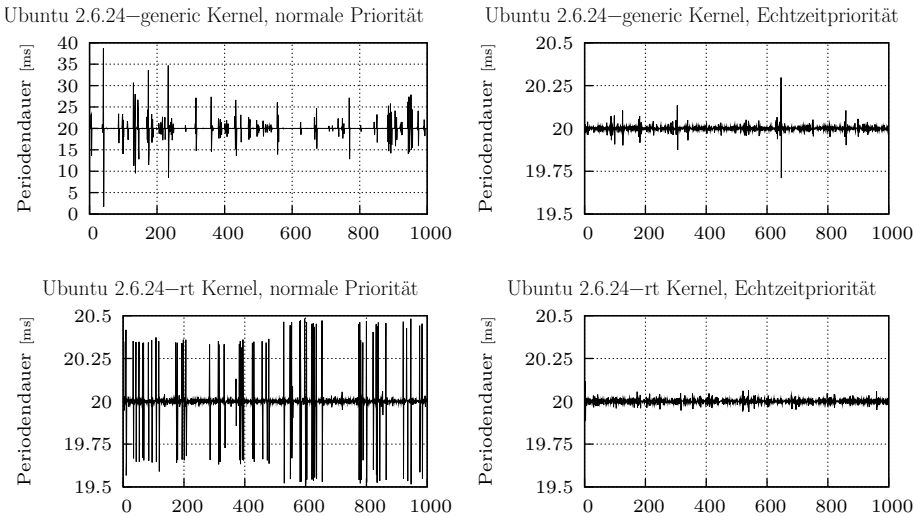
Da die maximal auftretende Latenz von besonderem Interesse ist, muss das System während der Zeitmessung ausgelastet sein. Es gibt viele Wege dies zu erreichen; hier wird eine 8-fach parallele Kernelkompilation verwendet. Dies lastet nicht nur alle CPUs aus, sondern generiert auch viele Interrupts von asynchronen I/O-Operationen, da eine Kernelkompilation viele Schreib- und Lesezugriffe auf der Festplatte verursacht. Weiterhin wird der Linux-Kern und der Prozess-Scheduler durch ständig neu erstellte Tasks, die es zu verwalten gilt, relativ stark belastet.

Die Quellen des aktuell laufenden Linux-Kernels erhält man mit diesem Befehl:

```
$ apt-get source linux-image-$(uname -r)
```

Danach finden sich die Quellen in einem Unterverzeichnis zum aktuellen Verzeichnis, und man kann die Erstellung des Linux-Kernels mit den nachfolgenden Befehlen starten. Diese sind vor jeder Messung in einer anderen Konsole im Hintergrund auszuführen:

```
$ cd linux-2.6.??
$ make clean
$ cp /boot/config-$(uname -r) .config
$ make -j8 bzImage
```



**Abb. 12.7.** Grafische Darstellung der Periodendauer von 1 000 aufeinanderfolgenden Ausführungen des `ServoThreads` für die vier untersuchten Konfigurationen.

Konfiguration	Min	Avg	Max
Ubuntu 2.6.24-generic Kernel, normale Priorität	0,01 $\mu$ s	24,71 $\mu$ s	18695,85 $\mu$ s
Ubuntu 2.6.24-generic Kernel, Echtzeitpriorität	0,01 $\mu$ s	9,35 $\mu$ s	297,90 $\mu$ s
Ubuntu 2.6.24-rt Kernel, normale Priorität	0,0 $\mu$ s	6,56 $\mu$ s	484,37 $\mu$ s
Ubuntu 2.6.24-rt Kernel, Echtzeitpriorität	0,01 $\mu$ s	6,78 $\mu$ s	116,88 $\mu$ s

**Tabelle 12.2.** Die minimale (Min), durchschnittliche (Avg) und maximale (Max) Abweichung von der Periode für die vier untersuchten Konfigurationen bei jeweils 1 000 aufeinanderfolgenden Ausführungen.

Abbildung 12.7 und Tabelle 12.2 zeigen die Ergebnisse der Benchmarks: In allen Konfigurationen beträgt die minimale Abweichung praktisch Null. Das bedeutet aber nur, dass das System in einer der 1 000 Perioden in der Lage war, die Zeitbedingung genau einzuhalten. Die durchschnittliche Abweichung

ist aussagekräftiger: Sie zeigt, dass Linux in allen Konfigurationen in der Lage ist, die Periode im Durchschnitt sehr gut einzuhalten, auch wenn das System durch die Kernelkompilation ausgelastet ist. Auch die beim 2.6.24-generic-Kernel mit normaler Priorität gemessenen 25  $\mu$ s sind bei der Ansteuerung des Servos vernachlässigbar. Allein die maximale Abweichung zeigt in den untersuchten Konfigurationen gravierende Unterschiede: Der 2.6.24-generic-Kernel leistet sich hier einen Ausreißer von ca. 18,5 ms, was fast einer ganzen Periode entspricht. Die anderen Konfigurationen erreichen auch hier passable Werte, wobei allerdings eine Abweichung von 0,5 ms (2.6.24-rt, normale Priorität) und 0,25 ms (2.6.24-generic, Echtzeitpriorität) durchaus kleine, aber spürbare Ruckler am Servo auslösen.

Betrachtet man die Ausreißer in Abbildung 12.7 genauer, so fällt auf, dass sie bei normaler Priorität in Bündeln aufzutreten scheinen. Wird die Periode einmal um einen größeren Wert verfehlt, so ist die Wahrscheinlichkeit höher, dass auch die nächste Periode verfehlt wird.

Zusammenfassend lässt sich daraus schließen, dass Linux selbst mit einem nicht auf Echtzeit optimierten Linux-Kernel mithilfe der Posix-Echtzeiterweiterung für weiche Echtzeitanwendungen nutzbar wird. Mit dem 2.6.24-rt-Kernel ist das Testsystem sogar in der Lage, während der durchaus als Extremlast einzustufenden parallelen Kernelkompilation noch ein sinnvolles Steuersignal an den Servo zu senden. Uneingeschränkt lässt sich dieser Benchmark allerdings nicht auf andere Systeme übertragen, da die maximale Latenz durch unterschiedliche Timerauflösungen und abweichende Konstellationen der geladenen Kernel-Module stark variieren kann. Entsprechend sind eigene Tests auf der anvisierten Zielplattform auf jeden Fall erforderlich.

Unterstützt die Zielplattform High-Resolution-Timer und benötigt die Anwendung bessere maximale Latenzzeiten als die hier vorgestellten, so muss man den zur Zeit noch aufwändigen Weg einschlagen und eine Echtzeiterweiterung wie Xenomai (siehe Tabelle 12.1) installieren. Hierdurch wird die Anwendung dann *hart* echtzeitfähig.

## Netzwerkkommunikation

### 13.1 Einführung

Fast alle Embedded-Linux-Boards enthalten mittlerweile eine Ethernet-Steckverbindung samt Netzwerk-Controller, die über das Betriebssystem leicht verwendet werden kann. Im Folgenden wird davon ausgegangen, dass ein Linux-Betriebssystem installiert, der Netzwerktreiber geladen ist und die Netzwerkschnittstellen verwendet werden können. Ein Aufruf von `ifconfig` auf der Konsole sollte außer *localhost* mindestens einen weiteren Eintrag liefern, um bspw. über `eth0` mit einem anderen System kommunizieren zu können:

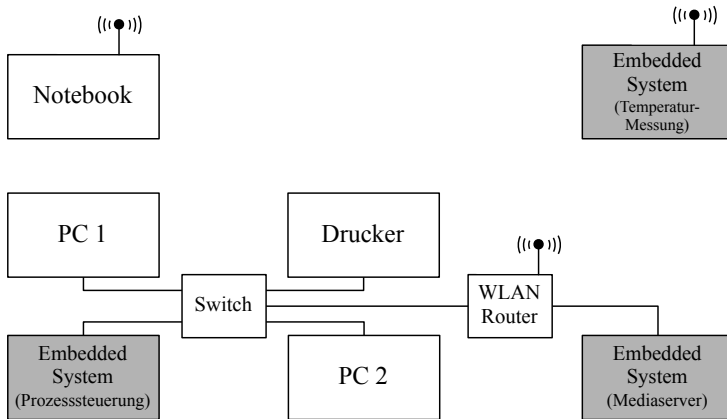
```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:18:39:36:xx:xx
          inet addr:192.168.1.160  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::218:39ff:fe36:xxxx/64  Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:12175227  errors:0  dropped:0  overruns:0  frame:0
          TX packets:13245102  errors:0  dropped:0  overruns:0  carrier:0
          collisions:0 txqueuelen:100
          RX bytes:3684740209 (3.4 GiB)  TX bytes:3454844338 (3.2 GiB)
...
```

Abschnitt A.3.1 erklärt, wie unter Linux Netzwerkeinstellungen vorgenommen werden. Wird der Befehl `ifconfig` nicht erkannt, so fehlen wahrscheinlich die Nutzerrechte, hier sei auf Abschnitt A.4.5 verwiesen. Mithilfe sog. *Sockets*<sup>1</sup> kann über verschiedene Netzwerkprotokolle mit Prozessen auf anderen Rechnern im Netzwerk oder auch lokal kommuniziert werden. Dies ist auch eine relativ einfache Möglichkeit, Embedded-Boards mit Peripheriegeräten oder Standard-PCs zu koppeln (vgl. Abbildung 13.1). Die Übertragungsbandbreite der Verbindung ist mit 10–1 000 Mbit/s wesentlich höher als bei I<sup>2</sup>C, CAN oder Bluetooth und durch die Ethernet-Topologie auch sehr flexibel.

---

<sup>1</sup> Als *Socket* wird der Endpunkt einer Netzwerkkommunikation bezeichnet.

Weiterhin ist die Kommunikation über Sockets auch über WLAN möglich. Einzige Voraussetzung hierfür ist das Vorhandensein eines passenden Gerätetreibers und die Sichtbarkeit des Netzwerkgerätes in der `ifconfig`-Ausgabe.



**Abb. 13.1.** Beispielszenario: Anbindung von Embedded-Systemen an ein vorhandenes Büronetzwerk.

Zu Beginn sei angemerkt, dass hier lediglich die Verwendung von Sockets unter Linux behandelt wird. Der Gebrauch unter Windows unterscheidet sich davon geringfügig.<sup>2</sup> Abschnitt 13.2.1 gibt zunächst eine Einführung in die Grundlagen der Socket-Kommunikation, bevor in Abschnitt 13.2.2 die *Berkeley Socket Library* vorgestellt wird, welche die Basis für fast alle Socket-Implementierungen bildet. Es folgt ein Programmbeispiel unter Verwendung der Berkeley Sockets in C, bevor mit den *Practical Sockets* in Abschnitt 13.2.6 eine C++-Implementierung vorgestellt wird, welche die Berkeley Sockets kapselt. Einfache Beispiele veranschaulichen die Verwendung dieser Bibliothek und zeigen in Abschnitt 13.2.7, wie eigene Protokolle auf Anwendungsebene verwendet werden können.

Um die Vorteile einer grafischen Benutzeroberfläche auf dem Host nutzen zu können, wird in Abschnitt 13.3 das Qt-Framework verwendet. Qt bietet eigene Schnittstellen zur Socket-Kommunikation und zum Multi-Threading, die an dieser Stelle erläutert werden. Die asynchrone Datenverarbeitung auf mehreren Rechnersystemen erfordert eine gewisse Unabhängigkeit der Hauptprozesse von der Kommunikation, damit diese nicht blockiert wird. Hierfür ist die Unterteilung in sog. Threads notwendig. In Kapitel 12 wurden Threads als Möglichkeit der quasiparallelen Datenverarbeitung vorgestellt. Die Bibliotheken aus Kapitel 12 werden hier verwendet und als bekannt vorausgesetzt.

<sup>2</sup> Unter Windows wird die Bibliothek *Winsock* verwendet, welche in Teilen kompatibel zu Posix-Systemen ist.

Für Qt-Novizen vermittelt Anhang D Einstieg und Grundlagen in das Qt-Framework.

Zum Datenaustausch im Netzwerk gibt es neben der Socket-Kommunikation auch die Möglichkeit, die Daten unter Verwendung systemeigener Dienste bereitzustellen. Eine Variante ist ein Webserver, welcher z.B. Prozessdaten des Rechners anzeigt und eine Eingabemaske zur Benutzerinteraktion bereitstellt. Dem Thema der Kommunikation zwischen Prozess und Webserver und der Erstellung eigener Weboberflächen ist der Abschnitt 13.4 gewidmet. Die Programmbeispiele dieses Kapitels sind in den Verzeichnissen `<embedded-linux-dir>/examples/sockets/` und `<embedded-linux-dir>/examples/qt/` zu finden, die notwendigen Klassenbibliotheken in `<embedded-linux-dir>/src/tools/`.

## 13.2 Datenübertragung via UDP

### 13.2.1 Grundlagen zu Sockets

Sockets werden für die Kommunikation zwischen Prozessen über ein Netzwerk verwendet. Ein Socket bildet hierbei einen Kommunikationsendpunkt der bidirektionalen Verbindung und stellt eine plattformunabhängige Schnittstelle zwischen Anwendung und Betriebssystem dar.

### TCP/IP-Referenzmodell

Beim TCP/IP-Referenzmodell handelt es sich um ein Schichtenmodell mit fünf Ebenen, welches für die Kommunikation mittels der Internet-Protokoll-Familie TCP/IP entworfen wurde. Später wurde dieses Modell zum ISO/OSI-Modell mit sieben Schichten ausgebaut [Wikipedia 08], wobei bestimmte Schichten oder Gruppen den im TCP/IP-Modell vorhandenen Ebenen entsprechen (vgl. Abbildung 13.2). Dieses Modell ist die Grundlage für die heutige Kommunikation im Internet und an dieser Stelle hilfreich, um die für eine Socket-Kommunikation relevanten Ebenen zu verdeutlichen.

In der Anwendungsschicht erfolgt die Datenübertragung über bekannte Protokolle wie FTP und HTTP. Weiterhin lassen sich in dieser Schicht über Telnet oder SSH Konsolenverbindungen zur Fernsteuerung entfernter Rechner herstellen und über SMTP E-Mails übermitteln. Diese Schicht und ihre Protokolle werden üblicherweise von der Anwendungs-Software bereitgestellt und sind nicht im Betriebssystem enthalten. Eine Voraussetzung für die Umsetzung eines eigenen Protokolls in dieser Schicht ist das Vorhandensein von Sockets auf der darunterliegenden Ebene.

Schicht im TCP/IP Modell	im OSI-Modell	Protokoll
Anwendungsschicht	5..7	HTTP, FTP
Transportschicht	4	TCP, UDP
Internetschicht	3	IP
Netzwerkschicht	1..2	Ethernet, FDDI

**Abb. 13.2.** TCP/IP-Referenzmodell und Entsprechungen im ISO/OSI-Schichtenmodell.

In der Transportschicht wird die Verbindung von einem zum anderen Ende aufgebaut und verwaltet. Hierzu sind zwei Kommunikationsendpunkte (Sockets) notwendig. Ein Endpunkt wird durch eine Adresse und eine Port-Nummer beschrieben und kommuniziert üblicherweise über TCP<sup>3</sup> oder UDP<sup>4</sup>.

Die Internetschicht ist für die Übertragung und Weiterleitung (*Routing*) von Paketen höherer Schichten zuständig. Mittels IP-Adresse und Subnetzmaske sind Teilnehmer in Subnetz-Gruppen als logische Einheiten gegliedert. Über das IP-Protokoll erfolgt auf dieser Schicht die Adressierung einzelner Rechner, um Übertragungswege auszumachen und Pakete zuzustellen. Eine zuverlässige Übertragung kann auf dieser Ebene allerdings nicht garantiert werden. IP wird aktuell weitgehend mit 32-bit-Adressen als IPv4 betrieben, es steht aber aufgrund der Adressknappheit die Verwendung von IPv6 mit 128-bit-Adressen kurz vor der Einführung.

Für die Netzwerkschicht ist im TCP/IP-Referenzmodell kein dediziertes Protokoll vorgesehen. Vielmehr können verschiedene Techniken der Punkt-zu-Punkt-Verbindung zum Einsatz kommen, wie bspw. Ethernet (CSMA/CD<sup>5</sup>), FDDI<sup>6</sup> oder Wireless LAN.

<sup>3</sup> Abkürzung für *Transmission Control Protocol*.

<sup>4</sup> Abkürzung für *User Datagram Protocol*.

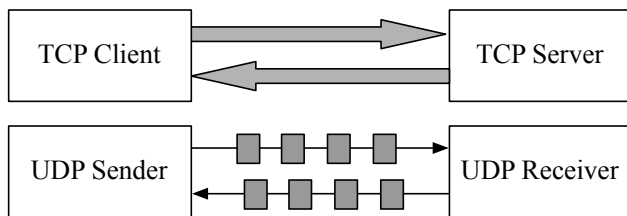
<sup>5</sup> *Carrier Sense Multiple Access and Collision Detection*. Übersetzt: *Mehrfachzugriff mit Trägerprüfung und Kollisionserkennung*. Standard-Zugriffsverfahren für Ethernet.

<sup>6</sup> *Fiber Distributed Data Interface*. Standard für eine auf Glasfaserkabel basierende Netzwerkarchitektur.

## TCP versus UDP

Für den Aufbau einer Socket-Verbindung muss der Anwender zunächst eines der beiden Protokolle TCP oder UDP auswählen, welche wesentliche Unterschiede aufweisen. Während TCP flussorientiert arbeitet, handelt es sich bei UDP um ein paketorientiertes Protokoll, bzw. um eine Übertragung einzelner Nachrichten oder Datagramme (siehe Abbildung 13.3).

TCP stellt einen virtuellen Kanal zwischen beiden Endpunkten her und zählt entsprechend zu den verbindungsorientierten Protokollen. Auf diesem Kanal ist eine bidirektionale Datenübertragung möglich, wobei Datenverluste erkannt und behoben werden. Das TCP-Protokoll ist somit ein zuverlässiges Verfahren der Datenübertragung, bei welchem Fehler in der Datenübertragung sicher erkannt werden. Aufgrund dieser Eigenschaften wird TCP als Grundlage für die meisten Protokolle auf Anwendungsschicht verwendet.



**Abb. 13.3.** Datenübertragung via TCP (flussorientiert) und via UDP (paketorientiert).

Bei UDP handelt es sich um ein verbindungsloses Protokoll, bei welchem einzelne Nachrichten, sog. Datagramme, übertragen werden. UDP gilt als nicht zuverlässiges Übertragungsverfahren, da es keine Garantie für eine erfolgreiche Übertragung gibt. Ebenso wenig wird garantiert, dass mehrere Pakete in der gleichen Reihenfolge und genau einmal ankommen. UDP verfügt allerdings über Prüfmechanismen, um sicherzustellen, dass ein ankommendes Paket auch die korrekten Daten enthält. Ein Datenaustausch via UDP ist aufgrund des einfacheren Protokolls und des Verzichtes auf einen Verbindungsaufbau im Vergleich zu TCP effizienter. Das unmittelbare Senden von Datagrammen bedeutet zudem eine geringere Latenz gegenüber einer gepufferten, flussorientierten Übertragung. Weiterhin lassen sich die Nachteile der Übertragungsunsicherheit durch bestimmte Mechanismen auf Anwendungsschicht wettmachen (Nummerierung der Nachrichten, zyklische Übertragung), sodass diese kaum mehr ins Gewicht fallen. Die Auswahl des Protokolls auf Anwendungsschicht ist grundsätzlich abhängig von der Aufgabe bzw. von den Anforderungen hinsichtlich der zu übertragenden Datenmengen, der maximalen Übertragungszeiten und der geforderten Sicherheit. Legt man eine Verbindung zwischen Embedded-System und PC zugrunde, über die zyklisch bspw.



Messwerte von etwa 250 Byte (50×32-bit-Variable + Paketinformation) alle 100 ms übertragen werden, so ist eine unverzügliche Übertragung der geringen Datenmenge wichtiger als eine hohe Datensicherheit. Würde bei einer TCP-Verbindung ein Paket verloren gehen, so würde dieses erneut angefordert.

Bis das angeforderte Paket einträfe, könnten bereits neue Daten vorliegen, womit die alten Daten bereits nicht mehr relevant wären. Diese Varianzen in der Übertragungsdauer disqualifizieren TCP als Protokoll für einen schnellen Prozessdatenaustausch (200 ms können hier als grober Richtwert dienen). Bei der Verwendung eines Standard-Linux-Betriebssystems und den entsprechenden Treibern ist UDP nicht echtzeitfähig im Sinne garantierter Übertragungszeiten. Es ist jedoch in jedem Fall schneller als TCP.

Als grobe Faustregel für eine Auswahl können folgende Kriterien gelten:

- TCP sollte für Aufgaben verwendet werden, bei denen ein hohes Maß an Datensicherheit erforderlich ist. Weiterhin ist es für die Übertragung kontinuierlicher Datenströme oder großer Datenmengen gut geeignet.
- Der Einsatz von UDP ist bei häufig wechselnden Übertragungspartnern und hohen Übertragungszyklen sinnvoll. Die Gewährleistung der Datensicherheit kann, falls gefordert, durch übergeordnete Protokolle erfolgen.

Im folgenden Kapitel wird für die Kommunikation zwischen Embedded-Systemen untereinander oder zwischen einem Embedded-System und einem PC ausschließlich UDP verwendet, allerdings immer mit den beschriebenen Restriktionen in Bezug auf Paketempfang und -reihenfolge. Da UDP selbst wiederum auf IP aufsetzt, begrenzt die dort verwendete maximale Größe auch die Größe der UDP-Datagramme auf maximal 65 535 Bytes. Hiervon entfallen 65 507 auf Nutzdaten-Bytes. Für tiefergehende Informationen sei an dieser Stelle auf das in der gleichen Reihe erschienene Buch *Grundlagen der Socket-Programmierung* hingewiesen [Zahn 06].

### 13.2.2 Berkeley Sockets

Die als *Berkeley Sockets* bekannte Schnittstelle wurde 1982 im Unix-Release 4.1cBSD<sup>7</sup> als Programmierschnittstelle in C implementiert. Mit dem Aufkommen von Netzwerkverbindungen war es das Ziel, ähnlich der IO-Routinen für den Dateizugriff auch eine Bibliothek für den Datenaustausch über das Netzwerk zu realisieren. Dies ist auch der Grund, weswegen die Socket-Schnittstelle bekannte Namen für die Zugriffsfunktionen wie `open()`, `read()`, `write()` oder `close()` verwendet.

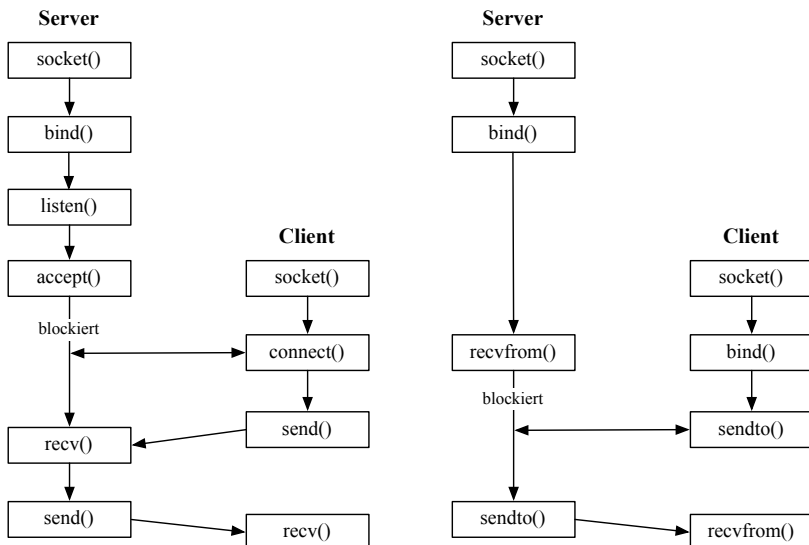
Die *Berkeley Socket Library* bündelt die Funktionen dieser Schnittstelle und ist in jeder Linux-Distribution integriert. Für die Verwendung müssen lediglich die

<sup>7</sup> *Berkeley Software Distribution* oder auch *Berkeley Unix*.

<code>&lt;sys/socket.h&gt;</code>	Für grundlegende Socket-Funktionen und Datenstrukturen.
<code>&lt;netinet/in.h&gt;</code>	Adressstrukturen für AF_INET- und AF_INET6-Adressfamilien, inklusive IP-Adresse mit TCP- und UDP-Port-Nummer.
<code>&lt;sys/un.h&gt;</code>	AF_UNIX-Adressfamilie. Nur für lokale Interprozesskommunikation, nicht für Netzwerke.
<code>&lt;arpa/inet.h&gt;</code>	Funktionen zur Umwandlung von numerischen IP-Adressen.
<code>&lt;netdb.h&gt;</code>	Funktionen zur Umwandlung von Protokoll- und Rechnernamen in Port-Nummern und IP-Adressen.

**Tabelle 13.1.** Wichtige Header-Dateien der Berkeley-Socket-Programmierschnittstelle und deren Bedeutung.

entsprechenden Header-Dateien inkludiert werden. Die relevanten Dateien sind in Tabelle 13.1 aufgelistet. Abbildung 13.4 zeigt die grundlegende Vorgehensweise zur Erzeugung einer verbindungsorientierten bzw. einer verbindungslosen Kommunikation. Während für eine verbindungsorientierte Übertragung mit den Befehlen `listen()`, `accept()` und `connect()` zunächst ein Kommunikationskanal aufgebaut werden muss, sind diese Funktionen für eine verbindungslose Kommunikation nicht relevant. Besteht eine Verbindung, so können Daten mittels `send()` und `recv()` kommuniziert werden. Für die verbindungslose Kommunikation ist es notwendig, beim Senden über `sendto()` zusätzlich die Empfängerdaten anzugeben. Beim Empfangen nimmt `recvfrom()` die Absenderinformation der Gegenstelle auf.



**Abb. 13.4.** Ablauf der Kommunikation über ein verbindungsorientiertes (links) bzw. ein verbindungsloses Protokoll (rechts) mit den beteiligten Funktionen der *Berkeley Socket Library*.

## Adressstrukturen

Die Adressstrukturen unterschiedlichen Typs werden der BerkeleySocketLibrary über Parameter übergeben. Dies kann teilweise etwas unübersichtlich sein, deshalb soll an dieser Stelle ein Überblick über verwendete Strukturen gegeben werden. Alle Funktionen der Bibliothek erwarten folgende, generische Adressstruktur:

```
#include <sys/socket.h>
struct sockaddr {
    u_short  sa_family;    // Adressfamilie
    char     sa_data[14];  // Protokollspezifische Adresse
};
```

**sa\_family** enthält die Adressfamilie und beschreibt die Art der Adressstruktur. Die eigentlichen Adressen werden in von **sa\_data** gespeichert. In **sys/socket.h** sind für die Adressfamilien Konstanten der Form **AF\_xxxx** definiert. Der Umgang mit dieser Struktur wäre relativ unhandlich, um darin bspw. die für eine Adressfamilie **AF\_INET** notwendigen Daten wie IP und Port-Nummer zu verwalten. Für jede Adressfamilie existiert deshalb ein angepasster Datentyp. Im Falle von **AF\_INET** ist dies die Struktur **sockaddr\_in**:

```
#include <netinet/in.h>
struct sockaddr_in {
    sa_family_t    sin_family;    // Adressfamilie
    in_port_t      sin_port;      // Port-Nummer
    struct in_addr  sin_addr;      // Internet-Adresse als 32-bit-Ganzzahl
    char           sin_zero[8];    // nicht verwendet
};
```

Zu beachten ist dabei, dass sowohl die IP-Adresse als auch die Port-Nummer in *Network Byte Order* angegeben werden (vgl. Abschnitt 13.2.5). Die jeweilige spezifische Struktur kann in die generische Struktur gecastet werden. Der Inhalt in **sa\_data** wird dann mithilfe von **sa\_family** richtig interpretiert. Entsprechend sind beim Aufruf der Socket-Funktionen in der Regel folgende Umwandlungen notwendig:

```
struct sockaddr_in addr;
<function>(sock, (struct sockaddr*)&addr,...)
```

Die wichtigsten Befehle werden nachfolgend beschrieben, bevor dann im Anschluss die Programmierbeispiele folgen. Für eine ausführliche Dokumentation sei auf die Unix-Manpages verwiesen.<sup>8</sup> Alle beschriebenen Funktionen geben im Fehlerfall den Wert **-1** zurück. Eine genauere Information über den Fehlergrund kann durch Auswertung der globalen Fehlervariablen **errno** bzw. deren Interpretation durch **perror()** erfolgen.

<sup>8</sup> Die Dokumentation zu den einzelnen Funktionen ist durch einen Kommandozeilenaufbau von **man <funktionenname>** verfügbar.

## Wichtige Funktionen der *Berkeley Socket Library*

### *socket()* – Erzeugung eines neuen Sockets

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Diese Funktion legt einen neuen Socket an und liefert als Rückgabewert den Deskriptor, im Fehlerfall `-1`. Der Fehlerfall tritt bei Ressourcenknappheit, fehlerhaften Einstellungen oder falschen Parameterkombinationen ein. `domain` beschreibt den Einsatzzweck und damit die Protokollfamilie des Sockets. Beispiele sind die Unix-Domain-Sockets (`PF_UNIX`), Appletalk (`PF_APPLETALK`) und die Internet-Protokoll-Familie IPv6 (`PF_INET6`). Üblicherweise wird aber die TCP/IP-Familie IPv4 mit dem Makro `PF_INET` verwendet.

Mit dem Parameter `type` wird der Typ des Sockets festgelegt. Verwendet wird für `PF_INET` in der Regel nur `SOCK_STREAM` oder `SOCK_DGRAM`, um einen flussbasierten oder einen paketorientierten Socket zu erzeugen. Das Protokoll wird in `protocol` angegeben. Hier handelt es sich entsprechend dem Verbindungstyp üblicherweise um `IPPROTO_TCP` bzw. `IPPROTO_UDP`, es können aber auch andere Protokolle verwendet werden. Um das vom System für diese Protokollfamilie definierte Standardprotokoll zu verwenden, wird `protocol=0` gesetzt.

### *bind()* – Zuweisung einer lokalen Protokolladresse an einen Socket

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sock, const struct sockaddr* addr, socklen_t addrlen);
```

Der Socket wird über den Descriptor `sock` referenziert. `addr` übergibt einen Zeiger auf die lokale Adresse, an welche der Socket gebunden werden soll. Da es sich hierbei um eine protokollspezifische Adressstruktur handelt, muss der Pointer unbedingt in die generische Struktur `sockaddr` gecastet werden. Bei Verwendung der Adressfamilie `AF_INET` im Rahmen des Protokolls `PF_INET` wird ein Socket über IP- und Port-Nummer identifiziert. Dazu dient die Struktur `sockaddr_in`. Durch die Verallgemeinerung muss zusätzlich in `addrlen` die Länge der in der Struktur angegebenen Adresse mitgegeben werden.

### *listen()* – Umstellung eines Socket auf den Lauschmodus

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sock, int backlog);
```

**sock** spezifiziert wiederum den Socket-Descriptor, **backlog** legt die maximal zulässige Anzahl an wartenden Verbindungswünschen fest. Verbindungswünsche werden vonseiten eines Clients durch den Aufruf von **connect()** initiiert und vom Server über **accept()** angenommen.

### ***accept()** – Annahme einer wartenden Verbindung*

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sock, struct sockaddr* addr, socklen_t* addrlen);
```

Ein mit **listen()** in den Lauschzustand versetzter Socket **sock** nimmt mit dieser Funktion wartende Verbindungen an. Nach dem Akzeptieren der Verbindung wird ein neuer Socket erzeugt und dessen Descriptor zurückgeliefert. Befinden sich zu diesem Zeitpunkt keine wartenden Verbindungen in der intern vorgehaltenen Liste, so blockiert die Funktion, bis eine Verbindung eintrifft. Um Informationen über den Endpunkt der Verbindung zu erhalten, kann ein Zeiger **addr** angegeben werden, welcher die Struktur der entsprechenden Adressfamilie referenziert. Auch hier muss wieder in die generische Struktur **struct sockaddr** gecastet werden. Entsprechend enthält **addrlen** einen Zeiger auf eine Variable, in welcher die Adresslänge des akzeptierten Endpunktes hinterlegt wird.

### ***connect()** – Verbindungsaufbau*

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sock, const struct sockaddr* servaddr, socklen_t addrlen);
```

Über **connect()** teilt der Client-Prozess einen Verbindungswunsch unter Verwendung seines lokalen Sockets **sock** mit. Die Funktion liefert als Rückgabewert 0, sobald die Verbindung serverseitig akzeptiert wurde. **servaddr** zeigt dabei auf die Adresse des Zielpunktes. **addrlen** gibt die Länge der referenzierten Adresse an.

### ***close()** – Schließen eines Sockets*

```
#include <unistd.h>
int close(int sock);
```

Wie jeder Datei-Descriptor, so sollte auch ein Socket so bald als möglich freigegeben werden und nicht erst implizit durch Beendigung des Prozesses. Das Verhalten von **close()** im Umgang mit aktiven Verbindungen kann über die Option **SO\_LINGER** festgelegt werden. Für das Setzen und Lesen der Socket-Optionen stehen **getsockopt()** und **setsockopt()** zur Verfügung. Die Unix-Manpages liefern hierzu weitere Informationen.

*recv(), recvfrom() – Datenempfang von einem Socket*

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int sock, void* buf, size_t len, int flags);
ssize_t recvfrom(int sock, void* buf, size_t len, int flags, struct
    sockaddr* addrfrom, socklen_t* addrfromlen);
```

Mit diesen beiden Funktionen werden Daten von einem Socket `sock` gelesen, welcher verbunden (`recv`) oder nicht verbunden (`recvfrom`) ist. `buf` zeigt auf einen Puffer, der die empfangenen Daten aufnehmen soll. `len` gibt abhängig von den gesetzten Flags die Größe des Puffers oder die Anzahl zu lesender Bytes an. Als Flags können unter Unix folgende Werte gesetzt werden:

- `MSG_PEEK`
- `MSG_OOB`
- `MSG_WAITALL`

Mit der Option `MSG_PEEK` wird lediglich in den Empfangspuffer hineingeschaut, die Daten aber nicht gelöscht. Dieses Ausleseverfahren eignet sich z. B., um den Header eines Datagramms mit variierendem Datenteil zu lesen, bevor die komplette Nachricht entnommen wird. `MSG_OOB` liest *Out-of-Band*-Daten (OOB), welche als dringend eingestuft werden und beim Senden entsprechend gekennzeichnet werden müssen. OOB-Daten werden mit höherer Priorität verschickt und normalerweise verwendet, um außergewöhnliche Ereignisse zu übertragen. Bei Sockets vom Typ `SOCK_STREAM` bewirkt `MSG_WAITALL` ein Blockieren von `recv()` so lange, bis die angegebene Anzahl Daten-Bytes empfangen wurde. Für nachrichtenorientierte Sockets blockiert die Funktion nur bis zum Eintreffen der nächsten Nachricht.

Im Erfolgsfall liefern beide Funktionen die Anzahl gelesener Bytes zurück, sonst `-1`. Die beiden zusätzlichen Parameter `addrfrom` und `addrfromlen` nehmen die Adresse der Quelle und deren Länge auf. Werden diese Werte auf `NULL` gesetzt, so wird die Information verworfen. `recvfrom()` wird meist in Verbindung mit UDP-Sockets eingesetzt, da hier in der Regel kein `connect()` durchgeführt wurde und `recv()` damit nicht möglich ist.

*send(), sendto() – Senden von Daten über einen Socket*

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sock, const void *msg, size_t len, int flags);
ssize_t sendto(int sock, const void *msg, size_t len, int flags, const
    struct sockaddr* addrto, socklen_t addrtolen);
```

Mit diesen beiden Funktionen werden Daten über einen Socket `sock` gesendet, die an der Stelle `msg` abgelegt wurden. `len` gibt dabei die zu sendende Anzahl Bytes an. Ähnlich wie bei den Empfangsroutinen können Flags wie

- MSG\_EOR
- MSG\_OOB

eingesetzt werden, um das Ende einer Nachricht zu markieren (MSG\_EOR) oder die Nachricht als dringlich zu kennzeichnen (MSG\_OOB). Die Markierung einer Ende-Nachricht ist wichtig, falls über das verbindungs- und paketerorientierte Protokoll SOCK\_SEQPACKET kommuniziert wird (es handelt sich hierbei um eine Mischform aus TCP und UDP). Für das Senden von Daten an nicht verbundene Empfänger mit `sendto()` ist zusätzlich die Angabe der Zieladresse und der Größe der Struktur notwendig. Im Erfolgsfall wird die Anzahl der versendeten Bytes zurückgegeben, sonst `-1`.

*getaddrinfo() – Bestimmung der Adressinformation aus einem DNS-Namen oder einer IP-Adresse*

```
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(const char *restrict nodename, const char *restrict
    servname, const struct addrinfo *restrict hints, struct addrinfo **
    restrict res);
```

Diese Funktion liefert Adressinformationen für einen Rechnernamen `nodename` und einen Dienst `servname` in Form von Strukturen des Typs `addrinfo`. Die Verwendung ist nicht protokollabhängig und ersetzt die älteren Funktionen `gethostbyname()` und `getservbyname()`. Allerdings ist diese Funktion in der Windows-Implementierung nicht enthalten und im Bezug auf einen portablen Quelltext nicht empfehlenswert.

Der Anwender hat die Möglichkeit, über `hints` einen Zeiger auf eine Adressstruktur mitzugeben, in welcher Informationen über Flags, Protokollfamilie, Socket-Typ und Protokoll enthalten sind. Alle anderen Angaben müssen 0 oder NULL sein. Über das letzte Argument `res` wird ein Zeiger auf eine verkettete Liste von Strukturen zurückgegeben, die mindestens einen Eintrag enthält. Die Struktur `addrinfo` ist folgendermaßen aufgebaut:

```
#include <netdb.h>
struct addrinfo {
    int      ai_flags;           // Flags
    int      ai_family;         // Adressfamilie
    int      ai_socktype;       // Socket-Typ
    int      ai_protocol;       // Protokoll
    size_t   ai_addrlen;        // Länge von ai_addr
    char *ai_canonname;         // Autorisierter Rechnername
    struct sockaddr *ai_addr;    // Binäre Adresse
    struct addrinfo *ai_next;    // Nächste Struktur in verketteter Liste
};
```

Das Pendant zu `getaddrinfo()` ist `getnameinfo()`, diese Funktion wird aber weit weniger oft benötigt.

BerkeleySockets stellen heutzutage als API einen Quasi-Standard für die Socket-Programmierung dar. Auch die unter Windows verwendete Winsock-Schnittstelle ist den BerkeleySockets nachempfunden. Nachdem die grundlegenden Funktionen erläutert wurden, soll die Berkeley Socket API im folgenden Beispiel für eine einfache UDP-Kommunikation in der Programmiersprache C verwendet werden. Die Bibliothek ist zudem Grundlage für weitere Implementierungen in C++, wie die in Abschnitt 13.2.6 vorgestellten *Practical Sockets*.

### 13.2.3 Verwendung der Berkeley Socket API

Im folgenden Beispiel sollen Daten von einer Sender-Anwendung (`udp_basic_sender`) über eine verbindungslose UDP-Kommunikation an ein Empfängerprogramm (`udp_basic_receiver`) übertragen werden. Es kommt dafür ausschließlich die in der Programmiersprache C erstellte BerkeleySocketLibrary zum Einsatz. Die beiden Anwendungen sind entsprechend ebenfalls in C programmiert. Zunächst wird die Datei `udp_basic_sender/sender.c` näher betrachtet:

```
#include <sys/types.h>      // Primitive system data types
#include <stdio.h>          // Input/Output
#include <stdlib.h>         // General utilities
#include <string.h>         // String handling
#include <sys/socket.h>     // Basic socket functions
#include <netdb.h>          // Translating protocol and host names

int main(int argc, char *argv[]) {
    struct addrinfo cfg,*srv;
    int fd, buflen;
    char buf[100];

    if (argc != 3) {        // Test for correct number of arguments
        printf("Usage: %s <Send IP> <Send Port>\n", argv[0]);
        exit(1);
    }
    printf("Sending UDP-packets to %s:%s..\n", argv[1], argv[2]);
    // make sure that defaults are 0/NULL
    memset(&cfg, 0, sizeof(struct addrinfo));
    cfg.ai_family   = PF_INET;
    cfg.ai_socktype = SOCK_DGRAM;
    cfg.ai_protocol = IPPROTO_UDP;
    if (getaddrinfo ( argv[1], argv[2], &cfg, &srv) != 0) {
        printf("Error resolving address\n");
        exit(1);
    }
    fd = socket(srv->ai_addr->sa_family, srv->ai_socktype, srv->ai_protocol);

    while(1) {
        printf("Enter your string now, ENTER to send\n");
        // read characters and send them
        scanf("%s", buf);
        buflen = strlen(buf) + 1;
        if ( sendto( fd, &buf, buflen, 0, srv -> ai_addr, srv -> ai_addrlen)
            != buflen) printf("Error sending string\n");
    }
    return 0;
}
```



Neben den Header-Dateien für Typdefinitionen und Standardfunktionen müssen zusätzlich `sys/socket.h` für Socket-Funktionen und `netdb.h` für Umwandlungsfunktionen eingebunden werden. Als Argumente werden der Anwendung die Zieladresse (oder der Host-Name) und der Zielport übergeben. Familie, Typ und Protokoll des Sockets werden vom Anwender in der Struktur `cfg` vom Typ `addrinfo` festgelegt. Um sich gegen Überraschungen abzusichern, empfiehlt es sich, diese zunächst mit `memset()` auf 0 (NULL) zu setzen. Die Funktion `getaddrinfo()` löst Host-Namen und Service auf und liefert als Ergebnis mit `srv` einen Zeiger auf eine Struktur vom Typ `addrinfo`, welche IP-Adresse und Port-Nummer enthält. Da es mehrere Verbindungswege zum Ziel geben kann, wird dieses Ergebnis in Form einer verketteten Liste zurückgeliefert. Der letzte Eintrag in `addrinfo` ist dabei ein Zeiger auf den nächsten Listeneintrag. Die vom Anwender in `cfg` spezifizierten Vorgaben werden in `srv` übernommen.

Wenn direkt eine gültige IP-Adresse angegeben wird, so ist dieser Zwischenschritt nicht unbedingt notwendig. Er erlaubt aber stattdessen auch die Verwendung eines Rechnernamens. Nachdem nun alle benötigten Informationen vorliegen, kann der Socket mit `socket()` erstellt werden.

Da die Kommunikation verbindungslos erfolgt, kann nun direkt mit `sendto()` unter Angabe von Zieladresse und -port gesendet werden. Hierbei ist zu beachten, dass bestimmte Ports für Systemdienste reserviert sind und nicht verwendet werden dürfen. Die Zuordnung wird von der Organisation IANA<sup>9</sup> verwaltet. Die Nummern 0–1023 sind den sog. *Well-Known-Ports* vorbehalten. Unter Unix werden Administratorrechte benötigt, um einen Port in diesem Bereich zu verwenden. Die Nummern 1024 bis 49151 zählen zu den *Registered Ports* und sollten für eigene Anwendungen ebenfalls nicht verwendet werden. So bleibt letztlich der Bereich von 49152 bis 65535 für die Nutzung in eigenen Programmen.

Die Anzahl der zu übertragenden Bytes ergibt sich aus der Länge der vorgegebenen Zeichenkette. Das Endezeichen wird ebenfalls übertragen. Im Erfolgsfall sollte die Länge der gesendeten Bytes wiederum `buflen` betragen. Auf der Gegenseite empfängt die Anwendung `udp_basic_receiver` eintreffende Daten. Ob ein Zuhörer existiert, spielt beim Versenden jedoch keine Rolle. Im Folgenden ist der Quelltext aus `udp_basic_receiver/receiver.c` dargestellt:

```
#include <sys/types.h>      // Primitive system data types
#include <stdio.h>          // Input/Output
#include <stdlib.h>         // General utilities
#include <string.h>         // String handling
#include <sys/socket.h>     // Basic socket functions
#include <netdb.h>         // Translating protocol and host names
#include <arpa/inet.h>      // Internet address manipulation

int main(int argc, char *argv[]) {

    struct addrinfo cfg,*srv;
```

<sup>9</sup> Internet Assigned Numbers Authority [IANA 08].

```

struct sockaddr_in client_addr;
socklen_t addrlen = sizeof(client_addr);
int fd, recvlen;
char buf[100];
char host[20];
char service[6];

if (argc != 2) {    // Test for correct number of arguments
    printf("Usage: %s <Receive Port>\n", argv[0]);
    exit(1);
}

printf("Receiving UDP-packets at port %s..\n", argv[1]);
// make sure defaults are 0/NULL
memset(&cfg, 0, sizeof(struct addrinfo));
cfg.ai_flags      = AI_PASSIVE;
cfg.ai_family     = PF_INET;
cfg.ai_socktype   = SOCK_DGRAM;
cfg.ai_protocol   = IPPROTO_UDP;
if (getaddrinfo ( NULL, argv[1], &cfg, &srv) != 0) {
    printf("Error resolving address or service\n");
    exit(1);
}
fd = socket(srv->ai_addr->sa_family, srv->ai_socktype, srv->ai_protocol);
if (bind ( fd, srv->ai_addr, srv->ai_addrlen) != 0) {
    printf("Error resolving address or service\n");
    exit(1);
}
while(1) {
    recvlen = recvfrom(fd, &buf, 100, 0, (struct sockaddr *) &client_addr,
        (socklen_t *) &addrlen);
    printf("Received string: %s from %s:%d\n", buf, inet_ntoa(client_addr.
        sin_addr), ntohs(client_addr.sin_port));
}

return 0;
}

```

Die Prozedur zur Erzeugung eines Sockets für den Datenempfang ist fast identisch zu vorigem Beispiel. Der kleine Unterschied ist, dass lediglich der Service, aber kein Host-Name mit `getaddrinfo()` aufgelöst werden muss. Nachdem der Socket mit `bind()` einer Port-Nummer zugeordnet wurde, kann auf diesem Port empfangen werden. Die in Form von Datenpaketen eintreffenden Zeichenketten werden empfangen und ausgegeben. Eine Angabe der maximalen Puffergröße von 100 Zeichen stellt sicher, dass `recvfrom()` nicht über den Puffer hinaus schreibt. Werden mehr Daten gesendet, so gehen diese verloren. Die Absenderinformationen werden in `client_addr` festgehalten.

Da es sich hier um die Protokollfamilie `PF_INET` handelt, und damit zwangsläufig um die Adressfamilie `AF_INET`, kann statt des abstrakten Datentyps `sockaddr` direkt die spezialisierte Version `sockaddr_in` verwendet werden, um Adressinformationen aufzunehmen. Über die Funktion `inet_ntoa()` wird die als 32-bit-Adresse vorliegende IP in die bekannte, punktierte Darstellung konvertiert und als String zurückgeliefert. Die Funktion `ntohs()` wandelt die Port-Nummer als 16-bit-Ganzzahl, welche im *Network Byte Format* vorliegt, bei Bedarf in die auf dem Zielsystem abweichende Darstellung um (siehe dazu Abschnitt 13.2.5). Wird nun eine Zeichenkette auf Senderseite

abgeschickt, so sollte diese unverzüglich beim Empfänger auftauchen. Voraussetzung hierfür ist natürlich, dass Adresse und Port-Nummer richtig gewählt wurden und dass eine physikalische Verbindung besteht. Übrigens handelt es sich bei der von der Empfänger-Anwendung angezeigten Nummer um die Port-Nummer, unter welcher die Nachricht abgesendet wurde. Diese hat mit dem Argument des Senders (Zielpport) nichts zu tun.

### 13.2.4 Socket-Debugging mit NetCat

Treten bei der Kommunikation mit Sockets Fehler auf, so hilft oftmals das Kommandozeilenprogramm NetCat weiter. Mit folgender Option können an Port 64000 eintreffende UDP-Daten auf der Kommandozeile dargestellt werden:

```
$ nc -l -u -v -p 64000
listening on [any] 64000 ...
```

Damit kann z. B. überprüft werden, ob die im vorigen Abschnitt erstellte Sende-Anwendung überhaupt Daten abschickt. Wird durch das Starten des Senders eine Verbindung erstellt, so wechselt **netcat** aus dem *Listen*- in den *Connect*-Modus. Nach einem wiederholten Start des sendenden Teilnehmers muss deshalb auch **netcat** neu gestartet werden, um die neue Verbindung zu akzeptieren.

Mit NetCat lassen sich auch Daten an einen Zielrechner senden:

```
$ echo "dies ist ein test" | nc -u -v 127.0.0.1 64000
```

Für eine detaillierte Beschreibung sei auf die Manpages von NetCat verwiesen. Unter <http://www.g-loaded.eu/2006/11/06/netcat-a-couple-of-useful-examples/> finden sich eine Vielzahl nützlicher Beispiele zum Umgang mit NetCat.

### 13.2.5 Host Byte Order und Network Byte Order

Abhängig von der verwendeten Architektur werden Ganzzahlen auf verschiedenen Systemen unterschiedlich im Speicher abgelegt. Ein Byte stellt dabei die kleinste adressierbare Einheit dar – die Repräsentation dafür ist auf allen Systemen gleich. Das Problem tritt bei Ganzzahlen auf, die mehr als ein Byte umfassen und entsprechend 16 oder 32 bit breit sind. Für die Darstellung von Fließkommazahlen, welche ebenfalls mehrere Bytes benötigen, stehen Standards zur Verfügung, wie diese im Speicher abzulegen sind (IEEE 754). Insofern besteht dieses Problem der unterschiedlichen Byte-Folgen zumindest in der Theorie nicht. In der Praxis halten sich jedoch nicht alle Chiphersteller daran, weshalb die Übertragung von Fließkommazahlen über das Netzwerk mit

Vorsicht zu genießen ist und besser vermieden werden sollte. Zwei Varianten kommen zum Einsatz, um Ganzzahlen im Speicher abzuliegen:

- Die **Big-Endian-First-Byte-Order** bzw. *Großes Ende zuerst*. Hier wird das Byte mit höchstwertigen Bits (d. h. die signifikantesten Stellen) an der kleinsten Speicheradresse hinterlegt, das Byte mit den niederwertigsten Bits an der größten Speicheradresse. Dieses Format wird unter anderem von Motorola-Prozessoren (PowerPCs, 68k) und Prozessoren der MIPS-Architektur verwendet.
- Bei der **Little-Endian-First-Byte-Order** bzw. *Kleines Ende zuerst* wird das Byte mit den niederwertigsten Bits (d. h. die am wenigsten signifikanten Stellen) an der kleinsten Speicheradresse gespeichert, das Byte mit den höchstwertigsten Bits an der größten Speicherstelle. Dieses Format wird bei Chips der Intel-Familie und damit entsprechend von allen x86-kompatiblen Prozessoren verwendet.

Aufgrund der historischen Herkunft werden diese beiden Formate auch als Motorola- bzw. Intel-Format bezeichnet. Für Prozessoren der ARM-Architektur ist das Format nicht fest vorgegeben. Diese Chips können zwischen *Little-Endian* und *Big-Endian* umgeschaltet werden.

Adresse	Big-Endian			Little-Endian		
	Hex	Dec	Char	Hex	Dec	Char
20 000	55	85	'U'	4E	78	'N'
20 001	4E	78	'N'	55	85	'U'
20 002	49	73	'I'	58	88	'X'
20 003	58	88	'X'	49	73	'I'

**Tabelle 13.2.** Little- und Big-Endian-Darstellung auf einem 16-bit-System.

Die Eigenschaft der unterschiedlichen Byte-Folgen ist auch als NUXI-Problem bekannt, da eine 32-bit-Ganzzahl, welche aus dem ASCII-Code der Zeichen „UNIX“ gebildet und auf einer Architektur mit Big-Endian-Darstellung abgelegt würde, auf einem Rechner mit Little-Endian-Architektur als „NUXI“ interpretiert würde (vgl. Tabelle 13.2). Um Daten zwischen Rechnern mit unterschiedlicher *Host Byte Order* kommunizieren zu können, hat man sich darauf geeinigt, für das Netzwerkformat bzw. für die *Network Byte Order* stets Big-Endian zu verwenden. Um eine Ganzzahl-Umwandlung zwischen der Darstellung im Netzwerk und des Hostrechners vorzunehmen, stehen in der Berkeley-Socket-Schnittstelle folgende Funktionen zur Verfügung:

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);           // host to network (long)
uint16_t htons(uint16_t hostshort);          // host to network (short)
uint32_t ntohl(uint32_t netlong);            // network to host (long)
uint16_t ntohs(uint16_t netshort);           // network to host (short)
```

Die Funktionen `htonl()` und `htons()` wandeln ganzzahlige Werte mit 32 oder 16 bit aus dem Format des Host-Rechners in das Netzwerkformat um. Die Funktionen `ntohl()` und `ntohs()` wiederum erzeugen aus ganzzahligen Werten mit 32 oder 16 bit des Netzwerkformats die passende Darstellung des Host-Systems. Socket-Parameter wie Port-Nummern oder 32-bit-Darstellungen von IP-Adressen sind immer in *Network Byte Order* anzugeben und werden von den Funktionen der Berkeley API auch so bereitgestellt. Werden eigene Protokolle auf Anwendungsschicht implementiert, wie in Abschnitt 13.2.8 gezeigt, so sollten die enthaltenen Daten ebenfalls einheitlich in *Network Byte Order* übertragen werden. Auch wenn diese Funktionen auf Maschinen mit Big-Endian-Architektur keine Auswirkung haben, so ist dies aus Gründen der Quellcode-Portabilität zu empfehlen. Das gilt gerade auch im Hinblick auf die Vielfalt möglicher Prozessorarchitekturen bei der Entwicklung für Embedded-Systeme.

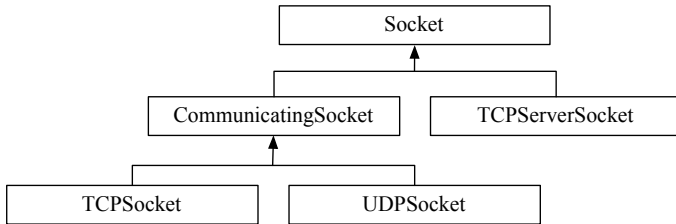
### 13.2.6 Practical Sockets

Die Verwendung der Berkeley Socket API wurde im vorigen Beispiel gezeigt. Die dortige Vorgehensweise zur Erstellung einer UDP-Verbindung passt jedoch nur schwer in ein objektorientiertes Programmierkonzept und ist darüber hinaus auch etwas umständlich. Mit der Bibliothek *Practical Sockets* von Michael J. Donahoo steht eine Sammlung von Wrapper-Klassen für die Berkeley Socket API zur Verfügung, um eigene TCP- und UDP-Verbindungen in Objektform zu erstellen [Donahoo 08]. Die Bibliothek besteht aus den Dateien `PracticalSockets.cpp` sowie `PracticalSockets.h` und ist im Verzeichnis `<embedded-linux-dir>/src/tools/` enthalten. Die Bibliothek ist der GNU General Public License unterstellt und darf in diesem Rahmen genutzt und verändert werden.

Abbildung 13.5 zeigt die Klassenhierarchie der *Practical Sockets*. Von einer abstrakten Klasse `Socket` wird spezialisiert, um entweder kommunizierende Sockets oder einen `TCPServerSocket` zu erhalten. Letzterer zeichnet sich dadurch aus, dass er selbst keine Kommunikation initiiert, sondern nur auf eingehende Verbindungen wartet und bei Bedarf einen kommunizierenden `TCPSocket` erzeugt. Als zweite Variante der kommunizierenden Sockets stehen `UDPSockets` zur Verfügung.

Die C++-Methoden orientieren sich bei der Namensgebung an den Funktionen der Berkeley API. Es soll deshalb an dieser Stelle nicht auf einzelne Funktionen eingegangen werden. Die Verwendung wird an einem Beispiel in Abschnitt 13.2.8 ausführlich erklärt. Die Bibliothek wird hier prinzipiell verwendet wie vom Autor vorgesehen, jedoch um folgende Funktion erweitert:

```
int UDPSocket::recvFromPeek(void *buffer, int bufferLen, string &
    sourceAddress, unsigned short &sourcePort) throw(SocketException);
```



**Abb. 13.5.** Klassenhierarchie der *PracticalSockets*.

Die Verwendung dieser Funktion anstelle von `recvFrom()` unterscheidet sich nur dahingehend, dass die gekapselte C-Funktion `recvfrom()` (vgl. Abschnitt 13.2.2) mit der Option `MSG_PEEK` aufgerufen wird. Damit werden Daten aus dem Empfangspuffer kopiert, ohne sie dort zu löschen.

### 13.2.7 Definition eigener Protokolle auf Anwendungsschicht

Mit den in *PracticalSockets* enthaltenen Sende- und Empfangsroutinen ist es möglich, beliebige Bytefolgen mit einer maximalen Länge von 65 507 Daten-Bytes zu übertragen. Der Inhalt kann komplexer Natur sein und sich z. B. aus einer Vielzahl von Variablen unterschiedlicher Datentypen zusammensetzen. Der Aufbau dieser Folge muss beiden Teilnehmern bekannt sein, weshalb es sich anbietet, ein gemeinsames Datenformat für UDP-Nachrichten zu definieren. Dies kommt einer Spezifikation des Protokolls auf Anwendungsschicht gleich. Je nach Anwendung kann es vorkommen, dass Datagramme unterschiedlichen Typs auf dem gleichen Socket (IP und Port) empfangen werden. In diesem Fall sollte eine Unterscheidung der angehängten Nutzdaten über einen einheitlichen Kopf bzw. Header erfolgen. Ein solches Datenformat könnte in Form einer Struktur wie folgt deklariert werden:

```

typedef struct {
    // header
    int id;
    int msg_count;
    int size;

    // data field
    int value1;
    short value2;
} datagram_1_t;

```

Im ersten Teil (Header) wird das Datagramm mithilfe einer ID eindeutig identifiziert. Zusätzlich kann eine fortlaufende Nummer sowie die Gesamtgröße des Datagramms enthalten sein. Dieser Kopf sollte für alle Datagramme gleich aufgebaut sein, danach folgt der individuelle Datenteil.

**Hinweis:** Es ist überaus ratsam, einer binär zu übertragenden Struktur eine konstante Größe zu geben. Das heißt konkret, dass keine Strings verwendet

werden dürfen, sondern stattdessen lediglich **char**-Arrays mit fester Größe. Auf Fließkommazahlen sollte aufgrund der teilweise vom Standard abweichenden Implementierungen ebenfalls verzichtet werden. Weiterhin ist es nicht sinnvoll, Zeiger mitzugeben, da diese auf dem Zielsystem kaum mehr von Nutzen sein werden. Sollte es unbedingt notwendig sein, in einem Datagramm Daten mit wechselnder Größe zu übertragen (z. B. ein Bild mit bestimmter Auflösung), so müssen die für eine korrekte Umwandlung notwendigen Informationen (Breite, Höhe und Farbtiefe des Bildes) im Header des Datagramms hinterlegt sein.

Diese Definitionen könnten in einer **.h**-Datei hinterlegt und damit verschiedenen Anwendungen zugänglich gemacht werden. Weiterhin sind jedoch Funktionen wünschenswert, um diese Daten zu manipulieren und bspw. eine Konvertierung in die *Network Byte Order* vorzunehmen (vgl. Abschnitt 13.2.5). Diese Manipulationen wiederum sind für den Header-Teil stets gleich, für den Datenteil jedoch unterschiedlich. Die Definition einer abstrakten Datagramm-Klasse **AbstractDatagram** liegt nahe, um die gemeinsamen Elemente und Methoden zu implementieren. Der Datenteil lässt sich in einer davon abgeleiteten Klasse beschreiben. Eine solche abstrakte Klasse könnte aussehen wie folgt:

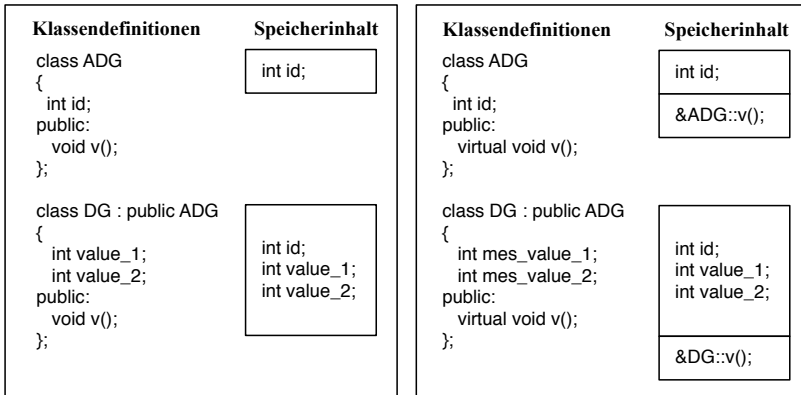
```
class AbstractDatagram {
public:
    AbstractDatagram(int id, int size);
    virtual ~AbstractDatagram();

    virtual void ntohData() = 0;
    virtual void htonData() = 0;
    void ntohHeader();
    void htonHeader();

private:
    int id;
    int msg_count;
    int size;
};
```

Die Konvertierungsmethoden der Nutzdaten müssen für die abgeleitete Klasse unbedingt implementiert werden. Eine Instanz der abgeleiteten Klasse könnte nun mit Verweis auf die Adresse und Größe mit **sendto()** verschickt und auf der Gegenseite empfangen werden. Leider funktioniert dies nicht direkt, da bei der Serialisierung und der Kopie der empfangenen Daten in ein existierendes Objekt die Informationen über virtuelle Funktionen<sup>10</sup> überschrieben werden. Dies hätte zur Folge, dass auf Empfängerseite die Aufrufe von **obj.ntohData()** oder **obj.htonData()** fehlschlagen würden, da die Referenzen auf die virtuellen Funktionen fehlerhaft sind. Abbildung 13.6 veranschaulicht das Problem. Dies soll zeigen, dass die Serialisierung von kompletten C++-Objekten mit Vorsicht zu behandeln ist und möglichst vermieden werden sollte.

<sup>10</sup> Als **virtual** deklarierte Funktionen können in einer abgeleiteten Klasse neu implementiert werden und „überschreiben“ die Methode der Basisklasse. Neben dem Überladen von Funktionsnamen und Operatoren ist dies ein weiteres Primitiv des Polymorphismus in C++.



**Abb. 13.6.** Das linke Schaubild zeigt die Speicherdarstellung für zwei Objekte der Klassen **AbstractDatagram** (ADG) und **Datagram** (DG), wobei DG von ADG abgeleitet ist. Jedes Objekt benötigt Platz für die Variablen der eigenen und der vererbten Klasse. Dem Compiler ist bekannt, welche Methoden für das jeweilige Objekt aufzurufen sind. Eine Referenz auf die Methoden innerhalb des Objekts ist nicht notwendig. Werden einzelne Methoden als virtuell deklariert (rechts), so wird vom Compiler eine spezielle Tabelle **vtable** angelegt, welche die Adressen der virtuellen Funktionen enthält.

Um dennoch die Vorzüge einer Klassenimplementierung mit zugehörigen Methoden zu wahren, wird eine zusammenhängende Struktur in Form eines **C-struct** in die abgeleitete Klasse aufgenommen und nur diese Struktur selbst serialisiert und kommuniziert. Dies hat zudem den Vorteil, dass die Kompatibilität mit einem in Standard-C programmierten Kommunikationspartner gegeben ist. Die abstrakte Klassenbeschreibung `<embedded-linux-dir>/src/tools/AbstractDatagram.h` besitzt lediglich einen Zeiger auf die Struktur `datagram_header_t`, der von den Methoden zur Manipulation verwendet wird. Die Struktur selbst ist erst im Datagramm enthalten, das vom Anwender spezifiziert wird. Die Methoden `ntoh()` und `hton()` führen eine Konvertierung der Header- und Nutzdaten aus, sofern dies notwendig ist. Die Bytefolgen-Darstellung des Host-Systems wird beim Anlegen eines Objekts bestimmt und in der statischen Variablen `isLE` hinterlegt. Die Beschreibung der Klasse **AbstractDatagram** ist im Folgenden abgedruckt:

```
class AbstractDatagram {
protected:
  typedef struct {
    u_int16_t id;                // header id
    u_int16_t size;              // size of entire message
    u_int32_t count;             // consecutive number
    u_int8_t has_network_format; // current state of byte order format
  } __attribute__((packed)) datagram_header_t;

public:
```



```

AbstractDatagram();
virtual ~AbstractDatagram();
void initDatagram(datagram_header_t* h, u_int16_t s, int id);
static bool checkLittleEndian();

virtual void ntohsData() = 0;
virtual void htonsData() = 0;
static void ntohsHeader(datagram_header_t* h);
static void htonsHeader(datagram_header_t* h);
void ntohs();
void htons();

char*      getData() { return (char*)header_p; }
u_int16_t  getId()   { return header_p->id; }
u_int16_t  getSize();
u_int32_t  getCount() { return header_p->count; }
u_int8_t   hasNetworkFormat() { return header_p->has_network_format; }

void incCount() { header_p->count++; }

private:
static bool isLE;
bool isLittleEndian() { return isLE; }
datagram_header_t* header_p;
u_int16_t msg_size; // msg size, this is needed for send routine and
                    // does not need conversion, always host format!
};

```

**Hinweis:** Es ist darauf zu achten, dass die Compiler auf beiden Systemen das gleiche *Data Structure Alignment* verwenden und entsprechend die jeweiligen Datentypen innerhalb der Struktur immer gleich viel Platz beanspruchen – gleichgültig, für welches Zielsystem sie übersetzt werden. Ist dies nicht sichergestellt, so lassen sich die Binärdaten nicht zwischen diesen Systemen austauschen. Üblicherweise wird in 32-bit-Systemen in Einheiten von vier Byte auf den Speicher zugegriffen. Felder in Strukturen werden deshalb standardmäßig auf ein Vielfaches dieser Größe gepackt.

Eine identische Repräsentation im Speicher wird einerseits durch die Verwendung der in `sys/types.h` definierten, plattformunabhängigen Datentypen erreicht, andererseits durch die Angabe des Attributs `packed` für die Typdefinitionen der Strukturen. Dadurch werden die einzelnen Feldgrößen auf die unbedingt benötigte Anzahl Bytes begrenzt und das Einsetzen von Füllbytes (Engl. *Padding*) durch den Compiler wird vermieden. Kleine Performanz-Einbußen werden in Kauf genommen, dafür wird die identische Darstellung auf 16- oder 32-bit-Systemen garantiert.

Die Definition eines Datagramms kann ähnlich wie in Beispiel `udp.cpp_sender/Datagrams.h` erfolgen. Die Nutzdaten werden hier in einer eigenen Struktur `measure_data_t` definiert, welche im Objekt direkt nach der Kopfstruktur `datagram_header_t` abgelegt wird. Beide Typdefinitionen der Strukturen sind mit dem Attribut `packed` versehen. Über den Aufzählungstyp `eDatagram` lässt sich der Typ des Datagramms eindeutig identifizieren. Diese Information wird, zusammen mit dem Zeiger auf die Kombination von Header- und Nutzdaten sowie deren Größe, im Konstruktor

durch den Aufruf von `initDatagram()` an das Vater-Objekt übermittelt. Die virtuellen Funktionen `htonData()` und `ntohData()` müssen von Anwender implementiert werden. Hierin enthalten sind die Aufrufe der Konvertierungsfunktionen aus der Berkeley-Socket-API. Die Definition eines eigenen Datagramms in `udp.cpp_sender/Datagrams.h` ist nachfolgend abgedruckt:

```
#include <linux/types.h>
#include "tools/AbstractDatagram.h"

enum eDatagram {
    MEASURE
};

class DatagramMeasure : public AbstractDatagram {
private:
    typedef struct {
        u_int16_t value1; // some sample values
        u_int32_t value2;
        char value3[20];
    } __attribute__((packed)) measure_data_t;
public:
    DatagramMeasure() : AbstractDatagram() {
        initDatagram(&header, sizeof(header) + sizeof(data), MEASURE);
    }
    virtual ~DatagramMeasure(){}

    virtual void htonData(){
        // implement data conversion here
        data.value1 = htons(data.value1);
        data.value2 = htonl(data.value2);
    }
    virtual void ntohData(){
        // implement data conversion here
        data.value1 = ntohs(data.value1);
        data.value2 = ntohl(data.value2);
    }

    // keep these together!
    datagram_header_t header;
    measure_data_t data;
};
```

Ein manueller Aufruf der Konvertierungsfunktionen `ntoh()` und `hton()` (nicht zu verwechseln mit den ähnlich lautenden Funktionen der Berkeley API) vor dem Senden und nach dem Empfangen stellt eine potenzielle Fehlerquelle dar. Es ist entsprechend ratsam, diese Funktionen direkt in die Sende- und Empfangsroutinen zu integrieren.

Die Klasse `UDPSocket` in `PracticalSocket.h` wurde deshalb um zusätzliche Funktionen erweitert, welche eine Referenz auf Daten vom Typ `AbstractDatagram` erhalten und falls notwendig Konvertierungen durchführen. Die Funktionen kommunizieren als Binärdaten lediglich die in `dgm` enthaltenen Header- und Datenstruktur(en):

```
void sendTo(AbstractDatagram* dgm, const string &
            foreignAddress, unsigned short foreignPort)
    throw(SocketException);
```

```
int recvFrom(AbstractDatagram *dgm, string &
             sourceAddress, unsigned short &sourcePort)
    throw(SocketException);
```

Eine Rückkonvertierung der Daten unmittelbar nach dem Senden sorgt dafür, dass die lokal vorgehaltenen Daten nun wieder regulär verwendet werden können. Der im Header eines jeden Datagramms hinterlegte Status `has_network_format` wird vor jeder Umwandlung überprüft, sodass auch ein manueller Aufruf von `ntoh()` oder `hton()` die Daten nicht versehentlich wieder in die falsche Darstellung konvertiert.

Es sei an dieser Stelle ausdrücklich darauf hingewiesen, dass diese Art der Implementierung von Datagrammen nicht thread-safe ist. Ein Zugriff auf die Membervariablen wäre sonst nicht direkt möglich, sondern nur mit viel Programmieraufwand und Performanz-Einbußen über Set- und Get-Funktionen. Es ist Aufgabe des Programmierers, die Datagramme und deren Strukturen je nach Aufgabe entsprechend abzusichern. Das Beispiel `qt/udp_qt_client` zeigt eine Möglichkeit dafür.

### 13.2.8 Verwendung der PracticalSockets

Eine einfache Sender-Empfänger-Kommunikation zeigt die Verwendung der PracticalSockets in den Beispielen `udp_cpp_sender,receiver`. Zunächst wird in der Datei `Datagrams.h` die Klasse `DatagramMeasure` als Container für Header- und Messdaten deklariert. Weiterhin sind dort die Konvertierungsfunktionen für den Datenteil implementiert. Im folgenden Quelltext des Senders ist nach dem Vorbelegen der Messdatenstruktur lediglich ein Socket vom Typ `UDPSocket` zu erstellen. Anschließend kann direkt über die Funktion `sendTo()` unter Angabe von Objektreferenz und Empfängeradresse gesendet werden. Der Zähler im Header wird dabei automatisch erhöht und Konvertierungen werden, falls notwendig, ebenfalls vorgenommen. Im Vergleich zum Listing aus Abschnitt 13.2.3 wird eine starke Vereinfachung bei mehr Funktionalität sichtbar. Im Folgenden ist der Quelltext der Datei `udp_cpp_sender/Sender.cpp` abgedruckt:

```
#include "tools/PracticalSocket.h" // UDPSocket and SocketException
#include <iostream>                 // cout and cerr
#include <cstring>                  // strcpy()
#include <cstdlib>                   // atoi()

#include "Datagrams.h"

using namespace std;

const int ECHOMAX = 255;           // Longest string to echo

int main(int argc, char *argv[]) {
    if (argc != 3) {               // Test for correct number of arguments
        cout << "Usage: " << argv[0] << " <Server> <Server Port>\n";
```

```

    exit(1);
}

DatagramMeasure mes;
mes.data.value1 = 0;
mes.data.value2 = 0;
strcpy(mes.data.value3, "Nachricht");

string serv_address = argv[1];           // server address
unsigned short serv_port = atoi(argv[2]); // server port

UDPSocket sock;

while(1) {

    // Send string to server
    sock.sendTo(&mes, serv_address, serv_port);
    cout << "Sending message no " << mes.getCount() << endl;

    mes.data.value1 = mes.getCount() % 2;
    mes.data.value2 = mes.getCount() % 10;
    sleep(1);
}
return 0;
}

```

Die im vorigen Quelltext erzeugten Daten werden durch das Programm **Receiver** empfangen und ausgegeben. Bei der Socket-Erstellung kann der Empfangs-Port mit angegeben werden. Ein Aufruf von `bind()` entfällt damit. `recvFrom()` empfängt die Strukturen aus `DatagramMeasure` und hinterlegt sie im entsprechenden Objekt. Zusätzlich werden die Absenderinformationen aufgefangen und ausgegeben. Die Anzahl der zu empfangenden Bytes wird automatisch aus dem Header des ankommenden Datenpaketes ermittelt, dann wird u. U. eine Umwandlung der Bytefolge vorgenommen. Das folgende Listing zeigt den Inhalt der Datei `udp_cpp_receiver/Receiver.cpp`:

```

#include "tools/PracticalSocket.h" // UDPSocket and SocketException
#include <iostream>                 // cout and cerr
#include <cstdlib>                  // atoi()

#include "../udp_cpp_sender/Datagrams.h"

const int ECHOMAX = 255;           // Longest string to echo

int main(int argc, char *argv[]) {

    if (argc != 2) {               // Test for correct number of arguments
        cout << "Usage: " << argv[0] << " <Server Port>" << endl;
        exit(1);
    }

    DatagramMeasure mes;
    unsigned short echoServPort = atoi(argv[1]); // Local port
    unsigned short sourcePort; // Source port
    string sourceAddress; // Source address

    UDPSocket sock(echoServPort);

    while (1) {
        sock.recvFrom(&mes, sourceAddress, sourcePort);
        cout << "Received packet from " << sourceAddress << ":"
            << sourcePort << " no " << mes.getCount() << endl;
    }
}

```

```

    cout << "Value 1: " << mes.data.value1 << endl;
    cout << "Value 2: " << mes.data.value2 << endl;
    cout << "Value 3: " << mes.data.value3 << endl << endl;
}
return 0;
}

```

Es handelt sich auch hier um eine unidirektionale Kommunikation, bei welcher die Empfangsfunktion blockiert, bis neue Daten eintreffen. Sollen Daten zusätzlich in die entgegengesetzte Richtung fließen, so ist die Verwendung weiterer Threads notwendig, wie das Beispiel zur bidirektionalen Kommunikation in Abschnitt 13.3.1 zeigt.

### 13.3 Kommunikation mit einer Qt-Anwendung

Qt ist ein objektorientiertes Framework zur Entwicklung plattformunabhängiger C++-Anwendungen für Linux, Windows und MacOS X. Neben Versionen für die genannten Plattformen steht mittlerweile ebenfalls eine Version für Embedded-Linux-Plattformen mit Framebuffer-Unterstützung zur Verfügung, vornehmlich zur Verwendung auf Mobiltelefonen. Für die in diesem Buch vorgestellten Geräte ist eine Verwendung dieser Embedded-Version wegen mangelnder grafischer Ausgabe bzw. fehlender Hardware-Unterstützung leider nicht möglich. Auf Qt für Embedded Linux soll deshalb an dieser Stelle nicht weiter eingegangen werden. Stattdessen wird die Linux-Variante für einen Ubuntu-basierten Standard-Desktop-PC verwendet. Auf dem PC lässt sich in einer Qt-Anwendung bspw. eine visuelle Aufbereitung von Messdaten oder ein Leitstand implementieren.

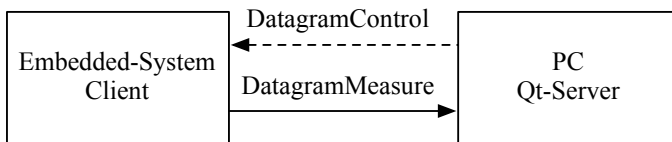
Neben der einfachen Erstellung von grafischen Oberflächen oder Konsolenprogrammen bietet Qt Unterstützungen für *Multithreaded Programming*, Netzwerkanbindung, OpenGL und XML. Die kommerzielle Version von Qt wird für viele bekannte Anwendungen wie Google Earth, Adobe Photoshop Elements und Skype verwendet. Durch die Plattformunabhängigkeit und die für Privatanwender freie Verfügbarkeit ist Qt auch im nicht-kommerziellen Bereich relativ verbreitet. Aus diesen Gründen ist die Wahrscheinlichkeit hoch, dass man im Lauf der Zeit mit Qt nochmals in Kontakt kommt. Eine Einarbeitung im Rahmen dieses Kapitels lohnt sich deshalb unbedingt. Für Qt-Novizen werden die grundlegenden Prinzipien in Anhang D.1 erklärt. Falls noch keine Erfahrung mit Qt vorhanden ist, so ist es ratsam, zuerst diese Einführung zu lesen. In Abschnitt 13.3.1 wird eine Kommunikation zwischen Qt-Anwendung und Embedded-Rechner erstellt und eine grafische Benutzeroberfläche implementiert.

Mit dem *Qt Designer* steht ein Werkzeug zur Verfügung, um grafische Oberflächen am Bildschirm aus verschiedenen Elementen per *Drag & Drop* zusam-

menzusetzen. Diese Elemente können in Form von *Qt-Designer-Plugins* selbst erstellt und wiederverwendet werden. Anhang D.2 gibt eine Einführung in Qt Designer und zeigt die Erstellung und Integration eigener Plugins. Das Beispiel einer Remote-Schrittmotorsteuerung in Abschnitt 13.3.2 demonstriert die Erstellung einer umfangreichen Benutzeroberfläche mit Qt Designer und die Verwendung externer Designer-Plugins.

### 13.3.1 Client-Server-Kommunikation mit Qt4

Im folgenden Beispiel soll ein bidirektionaler Datenaustausch zwischen einem Embedded-Client und einer PC-basierten Qt-Anwendung umgesetzt werden. Abbildung 13.7 zeigt den Zusammenhang. Die Quelltexte sind in den Verzeichnissen `qt/udp-qt.client` und `qt/udp-qt.server` hinterlegt.



**Abb. 13.7.** Bidirektionale Kommunikation zwischen einem Client auf einem Embedded-System und einem PC mit Qt-Server-Anwendung. Die Messdaten werden zyklisch übertragen, die Steuerdaten hingegen nur bei einer neuen Vorgabe.

Als anschauliche Aufgabe wird die Steuerung einer Positioniereinheit vorgegeben. Diese existiert nicht real, sondern wird vom Client simuliert. Über die Qt-Oberfläche wird eine Zielposition eingestellt, welche von der virtuellen Positioniereinheit angefahren wird. In bestimmten zeitlichen Abständen liefert der Client die aktuelle Position an den PC zurück. Die Deklaration der Datagramme in `udp-qt.server/Datagrams.h` (für beide Anwendungen im Verzeichnis des Servers) als Schnittstelle zwischen den Systemen sieht aus wie folgt:

```

#include <linux/types.h>
#include "tools/AbstractDatagram.h"

enum eDatagram {
    MEASURE,
    CONTROL,
};

class DatagramMeasure : public AbstractDatagram {
private:
    typedef struct {
        u_int16_t current_pos;
    } __attribute__((packed)) measure_data_t;
public:
    DatagramMeasure() : AbstractDatagram() {
        initDatagram(&header, sizeof(header) + sizeof(data), MEASURE);
    }
};
  
```

```

    }

    virtual void htonData(){
        // implement data conversion here
        data.current_pos = htons(data.current_pos);
    }
    virtual void ntohData(){
        // implement data conversion here
        data.current_pos = ntohs(data.current_pos);
    }

    // keep these together!
    datagram_header_t header;
    measure_data_t data;
};

class DatagramControl : public AbstractDatagram {
private:
    typedef struct {
        u_int16_t target_pos;
    } __attribute__((packed)) control_data_t;
public:
    DatagramControl() : AbstractDatagram() {
        initDatagram(&header, sizeof(header) + sizeof(data), CONTROL);
    }

    virtual void htonData(){
        // implement data conversion here
        data.target_pos = htons(data.target_pos);
    }
    virtual void ntohData(){
        // implement data conversion here
        data.target_pos = ntohs(data.target_pos);
    }

    // keep these together!
    datagram_header_t header;
    control_data_t data;
};

```

Zunächst wird die Client-Anwendung in `udp_qt_client/main.cpp` betrachtet. Um den Empfang der Daten unabhängig von der Verarbeitung und der Aktualisierung der aktuellen Position zu halten, wird der Empfang in einen separaten Thread `th` vom Typ `ReceiveThread` ausgelagert. Abgeleitet von der Klasse `Thread` werden im Konstruktor alle Daten übergeben, die dem Empfänger-Thread bekannt sein müssen. Dies sind die Datenstruktur `DatagramControl` mit einem Objekt vom Typ `Mutex`, um selbige zu schützen sowie der Empfangs-Port, an den der Server sendet. Wie bereits in Kapitel 12 erläutert, muss für jede von `Thread` abgeleitete Klasse die Methode `run()` überladen werden. In diesem Fall enthält `run()` letztendlich nur einen blockierenden Aufruf von `recvFrom()` in einer Endlosschleife. Nachfolgend ist die Implementierung der Klasse `ReceiveThread` aus dem Beispiel `udp_qt_client/main.cpp` abgedruckt:

```

class ReceiveThread : public Thread {
public:
    ReceiveThread(DatagramControl& control, Mutex& mutex_control, unsigned
        short listen_port)
        : data_control(control), mutex_data_control(mutex_control) {
        rec_sock.setLocalPort(listen_port);
    }

    virtual void run() {

        string source_address;           // Source address
        unsigned short source_port;      // Source port

        cout << "Receive Thread started" << endl;
        while(1) {

            int rec_size = rec_sock.recvFrom(&temp_data_control, source_address,
                source_port);
            try {
                cout << "Received new Target Position: " << temp_data_control.data
                    .target_pos << " from " << source_address << ":" <<
                    source_port << endl;
                MutexLocker ml(mutex_data_control);
                data_control = temp_data_control;
            }
            catch (SocketException) {
                cout << "Received exception " << endl;
                sleep(1);
            }
        }
    }

    ~ReceiveThread() {
        rec_sock.disconnect();
    }

private:
    UDPSocket rec_sock;
    DatagramControl temp_data_control;
    DatagramControl& data_control;
    Mutex& mutex_data_control;
};

```

Um die Gesamtdauer der Sperre von `data_control` kurz zu halten, wird diese Variable nicht während des blockierenden Aufrufs von `recvFrom()` geschützt, sondern es werden die Empfangsdaten zunächst in `temp_data_control` gespeichert und anschließend in die Zielstruktur kopiert. Adresse und Port der Server-Anwendung werden in `source_address` und `source_port` hinterlegt und ausgegeben.

Im zweiten Thread der Client-Anwendung, der eigentlichen `main()`-Methode, werden zunächst die Kommandozeilenargumente eingelesen und notwendige Datenstrukturen erzeugt und vorbelegt, bevor dann anschließend der Empfangs-Thread angelegt und gestartet werden kann. Die Endlosschleife errechnet zyklisch im Takt von `CYCLETIME` einen neuen Wert für die aktuelle Position `current_pos` und überträgt diesen anschließend zum Server. Um eine gewisse Trägheit zu simulieren, bewegt sich `current_pos` in jedem Durchlauf



um höchstens `STEP_DELTA` auf die Zielposition zu. Das folgende Listing zeigt die Implementierung der `main()`-Methode in `udp-qt-client/main.cpp`:

```
#include "../udp-qt_server/Datagrams.h"
#include "tools/PracticalSocket.h"
#include "tools/Thread.h"
#include "tools/Mutex.h"

#define CYCLETIME 200000 // Send data every 200 ms
#define STEP_DELTA 5 // steps per cycle towards target position
#define SIGN(a) ((a<0) ? -1 : 1)

class ReceiveThread : public Thread {
...
};

int main(int argc, char *argv[]) {

    if (argc != 4) { // Test for correct number of arguments
        cout <<"Usage: "<<argv[0]<<" <Server> <Server Port> <Receive Port>\n";
        exit(1);
    }

    string serv_address = argv[1]; // server address
    unsigned short serv_port = atoi(argv[2]);
    unsigned short listen_port = atoi(argv[3]);

    UDPSocket send_sock;
    DatagramMeasure mes;
    mes.data.current_pos = 0;
    Mutex mutex_control;
    DatagramControl control;
    control.data.target_pos = 0;

    ReceiveThread th(control, mutex_control, listen_port);
    th.start();

    while(1) {

        // compute new value
        mutex_control.lock();
        int diff = control.data.target_pos -- mes.data.current_pos;

        if ( abs(diff) < STEP_DELTA) {
            mes.data.current_pos = control.data.target_pos;
        }
        else {
            mes.data.current_pos += SIGN(diff) * STEP_DELTA;
        }
        mutex_control.unlock();

        // Send the current position to the server
        send_sock.sendTo(&mes, serv_address, serv_port);
        cout << "Sending message no " << mes.getCount() << endl;

        usleep(CYCLETIME);
    }
    send_sock.disconnect();

    th.wait();

    return 0;
}
```

Beim Blick auf den Server-Quelltext in `qt_udp_server` fällt auf, dass die `main()`-Methode wie in fast jeder Qt-Anwendung sehr kurz gehalten ist. Sie enthält lediglich die Instanziierungen der Objekten der Klassen `QApplication` und `ControlPanel`.

Die Klasse `ControlPanel` besitzt als Objektvariablen neben anderen Hilfsvariablen hauptsächlich zwei Sockets vom Qt-eigenen Typ `QUdpSocket`, Datencontainer für die zwei Datagramm-Typen und GUI-Elemente, welche innerhalb der Klasse bekannt sein müssen. Zwei als `private` deklarierte Slots dienen dem Abwickeln der Socket-Kommunikation. Die vollständige Klassenbeschreibung aus `udp_qt_server/ControlPanel.h` ist im Folgenden abgedruckt:

```
#include <QWidget>
#include <QSlider>
#include <QProgressBar>
#include <QUdpSocket>
#include "Datagrams.h"

class ControlPanel : public QWidget
{
    Q_OBJECT

public:
    ControlPanel(char *argv [], QWidget *parent = 0);

public slots:

private:
    QUdpSocket *udpSocketSend;
    QUdpSocket *udpSocketReceive;
    QHostAddress client_address;
    quint16 client_port;
    quint16 server_port;

    bool dataChanged;
    DatagramControl control_data;
    DatagramMeasure mes_data;
    QSlider *slider;
    QProgressBar *bar;

private slots:
    void broadcastDatagram();
    void processPendingDatagrams();
};
```

Das Objekt vom Typ `ControlPanel` integriert in diesem Fall die Benutzeroberfläche und die Socket-Verbindungen. Im Konstruktor wird zunächst die Benutzeroberfläche definiert und es werden Verbindungen zwischen Signalen und Slots erstellt. Ein sog. `QGridLayout` hilft, die Vielzahl von Elementen anzuordnen. In diesem rasterförmigen Layout werden dazu nacheinander die einzelnen *Widgets* unter Angabe der Position im Layout – und wahlweise auch der Ausdehnung – hinzugefügt und das Layout anschließend mit `setLayout()` zugewiesen.

Anschließend werden die beiden Sockets vom Typ `QUdpSocket` erzeugt und der Empfangs-Socket an den zugehörigen Port gebunden. Im Unterschied zum Client ist hier keine Auslagerung der Empfangsroutine in

einen separaten Thread notwendig, da der Empfang völlig transparent innerhalb Qt abläuft und der Socket nach dem Erhalt der Daten automatisch mit dem Signal `readyRead()` reagiert. Dieses Signal wird über einen `connect()`-Aufruf mit der Memberfunktion `processPendingDatagrams()` verbunden. Weitere Verbindungen dienen dem Beenden der Anwendung, dem Senden der Daten in `broadcastDatagram()` und der Kopplung der Anzeigeelemente für eine unmittelbare Aktualisierung. Die Kommunikation nach außen erfolgt in den beiden Memberfunktionen `broadcastDatagram()` und `processPendingDatagrams()`. Die Implementierung der Klasse `ControlPanel` mit zugehörigen Memberfunktionen sieht folgendermaßen aus (vgl. `udp-qt_server/ControlPanel.cpp`):

```
#include <QPushButton>
#include <QLCDNumber>
#include <QGridLayout>
#include <QApplication>
#include <QLabel>
#include <iostream>
#include "ControlPanel.h"

ControlPanel::ControlPanel(char *argv [], QWidget *parent)
: QWidget(parent)
{
    client_address = QHostAddress(argv[1]);
    client_port    = atoi(argv[2]);
    server_port    = atoi(argv[3]);

    QPushButton *button_quit = new QPushButton("Quit");
    QPushButton *button_send = new QPushButton("Send");
    QLabel *currpos_label = new QLabel("Current Position");
    QLCDNumber *lcd_currpos = new QLCDNumber();
    QLabel *targetpos_label = new QLabel("Target Position");
    QLCDNumber *lcd_targetpos = new QLCDNumber();

    slider = new QSlider();
    slider->setOrientation(Qt::Horizontal);
    bar = new QProgressBar();

    QGridLayout *layout = new QGridLayout();
    layout->addWidget(button_quit,1,0);
    layout->addWidget(button_send,3,0);
    layout->addWidget(currpos_label,0,1);
    layout->addWidget(targetpos_label,2,1);
    layout->addWidget(bar,1,1);
    layout->addWidget(slider,3,1);
    layout->addWidget(lcd_currpos,1,5);
    layout->addWidget(lcd_targetpos,3,5);

    setLayout(layout);

    udpSocketSend = new QUdpSocket(this);
    udpSocketReceive = new QUdpSocket(this);
    udpSocketReceive->bind(server_port);

    connect(button_quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    connect(button_send, SIGNAL(clicked()), this,
            SLOT(broadcastDatagram()));
    connect(udpSocketReceive, SIGNAL(readyRead()), this,
            SLOT(processPendingDatagrams()));
    connect(slider, SIGNAL(valueChanged(int)), lcd_targetpos,
            SLOT(display(int)));
    connect(bar, SIGNAL(valueChanged(int)), lcd_currpos,
```

```

        SLOT(display(int));
    }

    void ControlPanel::broadcastDatagram() {
        control_data.data.target_pos = slider->value();
        control_data.incCount();
        control_data.hton();
        udpSocketSend->writeDatagram(control_data.getData(), control_data.
            getSize(), client_address, client_port);
        control_data.ntoh();
        std::cout << "Send Target Value " <<
            control_data.data.target_pos << std::endl;
    }

    void ControlPanel::processPendingDatagrams()
    {
        while (udpSocketReceive->hasPendingDatagrams()) {

            udpSocketReceive->readDatagram(mes_data.getData(),
                mes_data.getSize());
            mes_data.ntoh();
            std::cout << "Received message no " << mes_data.getCount() << " with
                current value " << mes_data.data.current_pos << std::endl;
            bar->setValue(mes_data.data.current_pos);
        }
    }
}

```

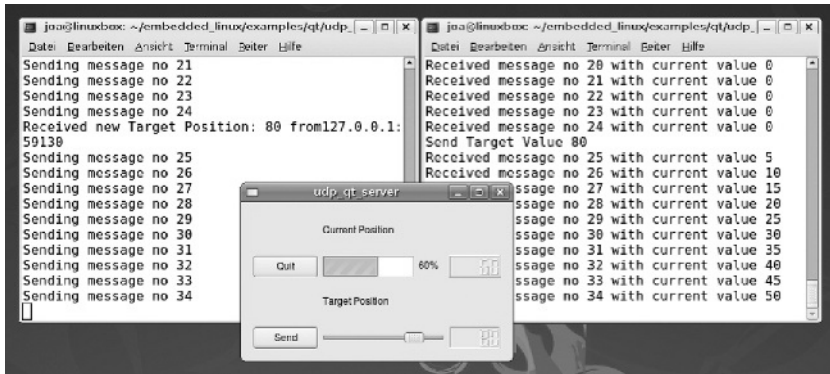
Die Funktionsaufrufe `writeDatagram()` und `readDatagram()` unterscheiden sich von den Practical-Socket-Routinen aus Abschnitt 13.2.8 dahingehend, dass als Argument ein `char`-Zeiger mit Angabe der Byte-Länge übergeben wird und keine Referenz auf ein Objekt vom Typ `AbstractDatagram`. Mittels der Funktion `getData()` ist deshalb ein Zeiger auf die eigentlichen Daten zu beziehen. Das bedeutet weiterhin, dass die notwendigen Konvertierungen vor und nach dem Senden sowie nach dem Empfangen der Daten manuell ausgeführt werden müssen. Einen weiteren kleinen Unterschied stellen die Qt-spezifischen Datentypen `QHostaddress` (IP-Adressen) und `quint16` (Port-Nummern) dar. Beim Erstellen eines Projektes mit Verwendung von `QUdpSockets` ist darauf zu achten, dass neben den Header- und Quelltextdateien in der Projektdatei `udp.qt_server.pro` folgende Zeile eingefügt wird:

```
QT += network
```

Werden Client und Server nun auf den jeweiligen Systemen mit den korrekten IP-Adressen und Port-Nummern als Argumenten gestartet, so sollten Konsolenmeldungen sowie eine grafische Ausgabe ähnlich wie in Abbildung 13.8 erscheinen.

### 13.3.2 Remote-Schrittmotorsteuerung mit grafischer Benutzeroberfläche

Kleinere grafische Oberflächen wie im ersten Beispiel können problemlos manuell erstellt werden. Im vorigen Beispiel wurde allerdings auch erkennbar, dass



**Abb. 13.8.** Steuerung einer virtuellen Positioniereinheit als Beispiel einer bidirektionalen Kommunikation zwischen Qt-Server und Client.

man damit bald an Grenzen stößt. Die Anordnung der Elemente mithilfe der `QGridLayouts` ist machbar, aber nicht sehr komfortabel. Hier können entsprechend einige Iterationen vergehen, bevor das Ergebnis aussieht wie gewünscht. Mit dem Qt Designer bietet *Trolltech* ein Werkzeug an, um grafische Oberflächen per *Drag & Drop* schnell und einfach zu erzeugen. In Anhang D.2 werden die Grundlagen für die Erstellung und Verwendung einer grafischen Oberfläche mithilfe des Qt Designers anhand einiger einfacher Beispiele vermittelt.

Am Beispiel der Ansteuerung eines Schrittmotortreibers TMC222 (vgl. Kapitel 9) über Client-Server-Anwendungen soll die Verwendung des Qt Designers zur Erstellung einer komplexen Benutzeroberfläche mit weiteren Plugins veranschaulicht werden.<sup>11</sup> Ziel ist die Programmierung eines Leitstandes, über den Zielvorgaben und Parametereinstellungen für den Motor erfolgen können und Statuswerte angezeigt werden. Die Ansteuerung und Verwaltung des I<sup>2</sup>C-Busses übernimmt ein Embedded-System wie bspw. die NSLU2. Der interessierte Leser findet in Anhang D.2.3 eine Anleitung zur Programmierung eigener Qt-Designer-Plugins.

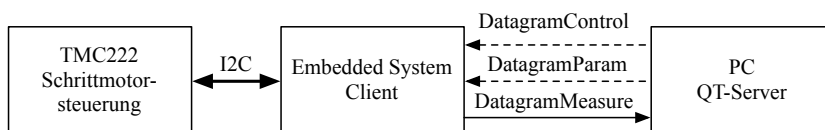
In der Client-Anwendung `udp_qtwidget_client` wird eine I<sup>2</sup>C-Schnittstelle benötigt, welche entweder direkt oder über einen IO-Warrior realisiert werden kann. Abschnitt 8.3 erklärt die verschiedenen Varianten. Weiterhin wird ein Schrittmotor-Modul TMC222 mit Motor verwendet, wie es bereits in Abschnitt 9.6 beschrieben wurde.

Für die Server-Anwendung `udp_qtwidget_server` muss sichergestellt sein, dass das Qt Designer Plugin *qLed* installiert ist. Falls noch nicht im Rahmen von Anhang D.2.2 geschehen, so kann dies durch einen Aufruf folgender Befehle nachgeholt werden:

<sup>11</sup> Die zugehörigen Beispiele finden sich in den Verzeichnissen `udp_qtwidget_client` und `udp_qtwidget_server`.

```
$ cd <embedded-linux-dir>/examples/qt/qt_plugins/qlcd
$ make
$ sudo make install
```

Abbildung 13.9 zeigt die Schnittstellendefinition zwischen Embedded-System und PC, welche auch hier in Form von Datagrammen in `udp.qtwidget_server/Datagrams.h` hinterlegt ist. Auf einen Abdruck wird an dieser Stelle aufgrund der Ähnlichkeit zum vorigen Beispiel verzichtet. Während das Embedded-System zyklisch Ist-Position und Statuswerte überträgt (`DatagramMeasure`), schickt der PC ereignisgesteuerte Nachrichten in Form von Reglervorgaben und Steuerungsparametern, die in `DatagramControl` und `DatagramParam` definiert sind. Eine Aufteilung in mehrere Datagramme ist immer dann sinnvoll, wenn die Daten aufgrund von *Events* oder unterschiedlichen Zykluszeiten ohnehin nicht synchron anfallen und bei einem umfangreichen Datagramm ein Großteil der Daten unnötigerweise übertragen würde.



**Abb. 13.9.** Steuerung und Überwachung eines am Embedded-System angeschlossenen Schrittmotors über eine Qt-Anwendung auf PC-Seite. Die Messdaten werden laufend aktualisiert, neue Regler- und Parametervorgaben hingegen ereignisgesteuert gesendet.

Während in `DatagramControl` mithilfe des Flags `new_target_pos` unterschieden werden muss, ob der Anwender am PC eine Referenzfahrt oder eine neue Zielfahrt ausführen möchte, zeigt bei der Übermittlung von `DatagramParam` die Variable `new_data` an, dass es sich um neue Parameter handelt. Dies scheint für das Parameter-Datagramm zunächst unnötig, da es sich bei dieser ereignisgesteuerten Nachricht immer um neue Parameter handelt. Die beiden Flags werden jedoch nicht nur zur Kommunikation zwischen PC und Embedded-System ausgewertet, sondern auch für die Abstimmung zwischen Empfangs-Thread und Haupt-Thread benötigt. Die Auswertung in `main()` erfolgt zyklisch, sodass Ereignisse über entsprechende Dateninhalte kommuniziert werden müssen. Um die Schnittstellen zwischen den Threads möglichst übersichtlich zu gestalten, bietet es sich an, die Variablen für die Inter-Thread-Kommunikation ebenfalls in die Strukturen zu integrieren.

Die Server-Anwendung unterscheidet sich vom vorigen Beispiel durch die Verwendung der Multi-Inheritance-Methode, um die Benutzeroberfläche einzubinden. Die Ableitung der Klasse `ControlPanel` von sowohl `QWidget` als auch `Ui::Panel` bewirkt, dass zwar sämtliche Anzeigeelemente in die Klasse `Ui_Panel` ausgelagert werden, aber dennoch direkt als Objektvariablen

verfügbar sind. Damit ergibt sich eine vergleichsweise übersichtliche Deklaration von `ControlPanel` (vgl. `udp_qtwidget_server/ControlPanel.h`):

```
#include <QWidget>
#include <QUdpSocket>
#include "Datagrams.h"
#include "iic/Stepper.h"
#include "ui_ControlPanel.h"

class ControlPanel : public QWidget, private Ui::Panel
{
    Q_OBJECT

public:
    ControlPanel(char *argv [], QWidget *parent = 0);
    virtual ~ControlPanel();

public slots:

signals:

private:

    QUdpSocket *udpSocketSend;
    QUdpSocket *udpSocketReceive;
    QHostAddress client_address;
    quint16 client_port;
    quint16 server_port;
    QTimer *timer;

    bool dataChanged;
    DatagramControl control_data;
    DatagramMeasure mes_data;
    DatagramParam param_data;
    int num_mesg_rec, num_mesg_send;

private slots:
    void broadcastDatagramParam();
    void broadcastDatagramControl();
    void sendNewTargetpos();
    void sendGotoRefpos();
    void processPendingDatagrams();
    void noConn();
};
```

Die Erweiterung um zusätzliche Slots erlaubt es, die Absendeknöpfe mit `connect()` direkt an die passende Verteilungsfunktion zu koppeln. Dies bedeutet, dass komplette Datagramme entweder über `broadcastDatagram<xy>`-Funktionen direkt gesendet werden oder zunächst in einem anderen Sende-Slot einzelne Datagramm-Elemente aufbereitet und erst dann übertragen werden. Der Großteil der Implementierung in `udp_qtwidget_server/ControlPanel.cpp` behandelt die Datenaufbereitung aus dem Formular und das Füllen der Datagramm-Strukturen. Der Quellcode dazu ist selbsterklärend, auf einen Abdruck wird deshalb an dieser Stelle verzichtet.

Die Client-Anwendung `udp_qtwidget_client` orientiert sich ebenfalls stark am vorigen Client-Beispiel `udp_qt_client`. Der Datenempfang wird in einen Thread `th` vom Typ `ReceiveThread` ausgelagert. Die Verarbeitung der Da-

ten findet in `main()` statt. Der Empfangs-Thread bekommt im Konstruktor alle notwendigen Daten bzw. Referenzen, inklusive der Mutexe, um diese zu schützen, geliefert.

Eine Besonderheit ergibt sich beim Empfang der Daten: Da nun zwei Datagramme auf dem gleichen Port ankommen, müssen diese voneinander unterschieden werden. Dies wird notwendig, um einerseits die korrekte Größe auszulesen und die Daten andererseits richtig abzulegen. Der Datagramm-Header erhält nun endlich seine Berechtigung, denn hier ist der jeweilige Typ im Feld `id` hinterlegt. Durch die Verwendung von `recvFromPeek()` ist es möglich, die Header-Daten lediglich zu kopieren, aber noch nicht aus dem Puffer zu löschen. So kann eine Unterscheidung je nach ankommendem Typ getroffen werden, bevor das gesamte Datagramm mit `recvFrom()` tatsächlich dem Puffer entnommen wird. Da sichergestellt ist, dass nun bereits Daten vorliegen, können diese umgehend in die jeweiligen Datagramm-Strukturen eingelesen werden. Die `run()`-Methode des Empfangs-Threads sieht damit folgendermaßen aus (vgl. Klasse `ReceiveThread` in `udp_qtwidget_client/main.cpp`):

```
void run() {

    string sourceAddress;      // Address of datagram source
    unsigned short sourcePort; // Port of datagram source

    cout << "Receive Thread started" << endl;
    while(1) {

        try {
            // do not take data from socket until you know what it is
            int rec_size = rec_sock.recvFromPeek(&header, sizeof(
                datagram_header_t), sourceAddress, sourcePort);
            AbstractDatagram::ntohHeader(&header);

            if (header.id == CONTROL) {
                MutexLocker ml(m_data_control);
                int rec_size = rec_sock.recvFrom(&data_control, sourceAddress,
                    sourcePort);
            }
            else if (header.id == PARAM) {
                MutexLocker ml(m_data_param);

                int rec_size = rec_sock.recvFrom(&data_param, sourceAddress,
                    sourcePort);
            }
        }
        catch (SocketException) {
            cout << "Received exception " << endl;
            sleep(1);
        }
    }
}
```

In der `main()`-Methode des Clients wird der I<sup>2</sup>C-Bus samt Stepper-Objekt erzeugt und die Parameterstruktur mit unkritischen Daten vorbelegt, um eine Initialisierung durchführen zu können. Nach dieser folgt eine Endlosschleife mit folgenden Aktionen:

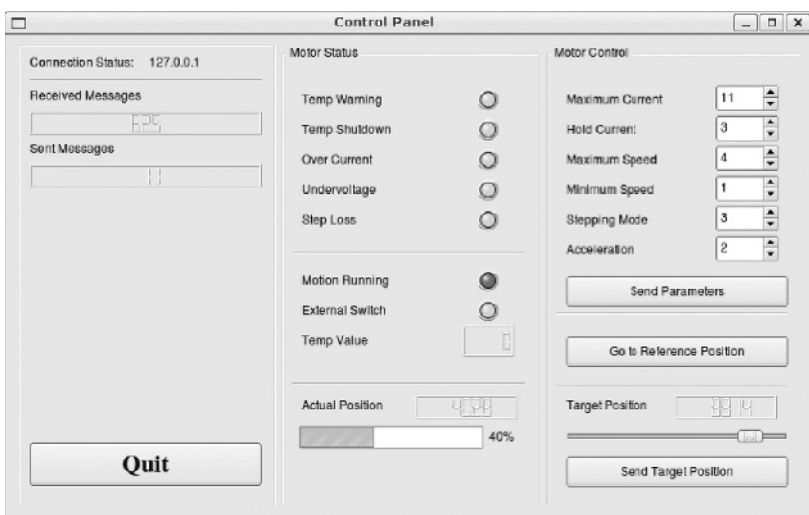
1. Abarbeitung neuer UDP-Daten,



2. Abhandlung der I<sup>2</sup>C-Übertragung, Senden von Befehlen, Statusabfragen,
3. Senden der UDP-Daten,
4. Warten auf einen neuen Zyklus.

Wird die Server-Anwendung übersetzt und gestartet, so erscheint eine Benutzeroberfläche ähnlich Abbildung 13.10. Ist dies nicht der Fall, dann wurde wahrscheinlich die Bibliothek `qLed` noch nicht installiert bzw. noch nicht in den Pfad eingetragen (vgl. Anhang D.2.2).

Treten weitere Fehler auf, so empfiehlt es sich, zunächst die Funktionalität der Schrittmotorsteuerung über das Stepper-Beispiel aus Abschnitt 9.6 zu überprüfen. Ob eine UDP-Kommunikation zwischen den Teilnehmern möglich ist, kann mithilfe der in den vorigen Abschnitten aufgelisteten UDP-Beispiele getestet werden.



**Abb. 13.10.** Qt-Leitstand zur Steuerung des Stepper-Moduls aus Abschnitt 9.6.

Anmerkung: Die Client-Anwendung muss nicht zwangsläufig auf einem Embedded-System laufen. Bei Verwendung eines IO-Warriors kann dieser auch direkt am Host-PC angesteckt werden. Hierbei zeigt sich ein weiterer Vorteil der Sockets: Die Steuerung kann – unabhängig von der Client-Anwendung – komplett PC-basiert lokal entwickelt und getestet werden.

## 13.4 Interaktion mit einem Webserver via CGI

In den vorigen Abschnitten dieses Kapitels wurde gezeigt, wie Daten von einem Embedded-System über das Netzwerk kommuniziert und auf einem anderen Rechner dargestellt oder weiterverarbeitet werden können. Dies setzt voraus, dass der Zielrechner entsprechend als Gegenstelle konfiguriert ist und dass die zur Kommunikation notwendigen Anwendungen dort lauffähig sind. Eine Möglichkeit, diese Forderungen zu umgehen und auf eine universelle Schnittstelle auszuweichen, ist die Verwendung eines Webserver. Auch auf Embedded-Systemen mit geringer Rechenleistung laufen herkömmliche Webserver mit passabler Geschwindigkeit, zumindest solange auf Skriptsprachen wie *PHP*<sup>12</sup> und umfangreiche Datenbanken wie *MySQL*<sup>13</sup> verzichtet wird. Auf einer selbsterstellten Website lassen sich Messwerte und Statusdaten anzeigen oder – bei entsprechender Gestaltung – auch Eingaben tätigen. Der Zugriff kann über einen herkömmlichen Webbrowser erfolgen und ist unabhängig vom verwendeten Betriebssystem. Komplex ist nur die Umsetzung der Interaktion zwischen der eigenen Anwendung und dem Webserver auf dem Embedded-System.

Im folgenden Abschnitt soll diese Funktionalität mithilfe des *Common Gateway Interface (CGI)* umgesetzt werden. Dabei werden verschiedene Möglichkeiten der Interaktion zwischen Anwendung und Webserver behandelt. Die relevanten Beispiele sind im Verzeichnis `<embedded-linux-dir>/examples/cgi/` zu finden. Grundlegendes Wissen über den Aufbau von HTML-Dateien wird in diesem Abschnitt vorausgesetzt – die SELFHTML-Website<sup>14</sup> bietet eine hervorragende Einführung in verschiedene Web-Technologien. Für diesen Abschnitt sind insbesondere die Bereiche *HTML/XHTML*, *JavaScript/DOM* und *Webserver/CGI* der SELFHTML-Website relevant.

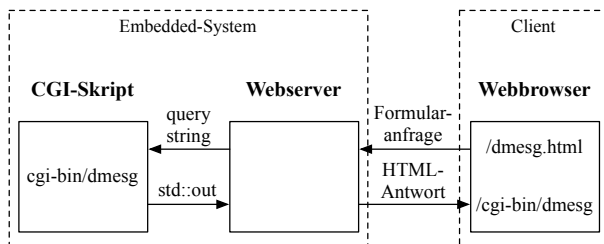
Über die CGI-Schnittstelle kann der Webserver Skripte oder kompilierte Programme aufrufen und das Ergebnis einem anfragenden Webbrowser als dynamischen Inhalt zurückliefern. In den Skripten oder Programmen finden interne Verarbeitungsschritte statt, wie z. B. das Aufsummieren eines Benutzer-Zählers, um aus diesen veränderlichen Daten dann wiederum dynamischen HTML-Code zu erzeugen. Oftmals werden Bash- oder Perl-Skripte in HTML-Dateien aufgerufen und auf dem System des Servers durch einen Interpreter ausgewertet. Dies ist in Hinblick auf die Portabilität zwischen Servern mit verschiedenen Betriebssystemen und Architekturen eine sichere Variante. Das Argument ist aber insofern vernachlässigbar, als im vorliegenden Fall davon ausgegangen wird, dass ein Rechenprozess als Eigenentwicklung auf

<sup>12</sup> Skriptsprache für die Webprogrammierung. Vgl. <http://www.php.net/>.

<sup>13</sup> Open-Source-Datenbank. Vgl. <http://www.mysql.de/>.

<sup>14</sup> Vgl. <http://de.selfhtml.org/>.

dem Embedded-System läuft und entsprechend auch problemlos ein CGI-Programm kompiliert werden kann.



**Abb. 13.11.** Kommunikation zwischen CGI-Skript, Webserver und -browser im Beispiel `cgi.basic`. Die Datenübermittlung an das Skript geschieht über die Umgebungsvariable `QUERY_STRING` (`GET`-Methode) oder über die Standardeingabe (`POST`-Methode). Die Antwortdaten werden in beiden Fällen über die Standardausgabe an den Webserver kommuniziert.

Abbildung 13.11 zeigt die Kommunikation zwischen dem Webserver und einer CGI-Anwendung. Diese kann in der HTML-Datei über verschiedene Mechanismen aufgerufen werden. Eine Möglichkeit ist die Verwendung eines Formulars, für das eine Aktion festgelegt wird:

```
<FORM ACTION="/cgi-bin/ps" METHOD="GET">
  <INPUT TYPE="submit" VALUE="ps">
</FORM>
```

Das Skript `/cgi-bin/ps` wird in diesem Fall durch ein Betätigen des mit `<input...>` definierten HTML-Knopfes aufgerufen. Dabei werden die Wertepaare des Formulars als Zeichenkette der Form `field1=value1&field2=value2...` übermittelt. Diese Daten werden auch als *query string* bezeichnet. Als Übertragungsmethode kommt `GET` oder `POST` zum Einsatz. In beiden Fällen werden die Daten in einem *query string* kodiert. Die Methoden unterscheiden sich folgendermaßen:

**GET** – Die Formulardaten werden in Form einer Umgebungsvariablen `QUERY_STRING` bereitgestellt. Der Datenstrom wird an die Adresse des CGI-Skripts angehängt und ist nach dem Absenden in der Adresszeile sichtbar.

**POST** – Bei dieser Methode wird der Datenstrom direkt an die CGI-Anwendung gesendet und muss von dieser über die Standardeingabe gelesen werden. In der Umgebungsvariablen `CONTENT_LENGTH` wird die Länge der Zeichenkette übermittelt, da am Ende kein Sonderzeichen folgt.

Üblicherweise wird vorab festgelegt, welche Schnittstelle verwendet werden soll. In der CGI-Anwendung kann dies durch entsprechende Programmierung aber auch automatisch festgestellt werden. Nach der Verarbeitung der Daten reagiert die CGI-Anwendung, indem sie auf die Standardausgabe antwortet.

Diese Schnittstelle ist unabhängig von der verwendeten Übertragungsmethode immer gleich.

Das Beispiel `cgi.basic` zeigt eine einfache Kommunikation über ein Bash-Skript. Vorher sind jedoch noch einige Vorkehrungen zu treffen: Zunächst muss sichergestellt sein, dass auf dem Embedded-System ein Webserver installiert ist. Sollte dies nicht der Fall sein, so wird die Installation unter Debian mit folgendem Befehl nachgeholt:<sup>15</sup>

```
$ sudo apt-get install apache2
```

Ein Aufruf der Adresse `http://<embedded-system-ip>` im Webbrowser eines anderen Rechners sollte eine Website *It works!* oder alternativ den OpenWrt-Anmeldebildschirm darstellen. Erscheint keine Website, so läuft wahrscheinlich der Webserver nicht (ggf. `sudo /etc/init.d/apache2 start` ausführen, bzw. `sudo /etc/init.d/httpd start` für OpenWrt).

In der Konfigurationsdatei<sup>16</sup> sind wichtige Angaben über die Verzeichnisse der HTML-Dateien und CGI-Skripte hinterlegt.<sup>17</sup> Der Webserver zeigt HTML-Dateien an, die im HTML-Wurzelverzeichnis `DocumentRoot` gefunden werden. Standardmäßig ist unter Debian `/var/www` die Vorgabe. Auf anderen Distributionen wird auch oft `/srv/www` oder `/opt/www` verwendet. Weiterhin können in der Standardkonfiguration von Apache nur CGI-Skripte ausgeführt werden, die im durch `ScriptAlias /cgi-bin/` spezifizierten Verzeichnis liegen. Üblicherweise ist dies `/usr/lib/cgi-bin/`. Um die mitgelieferten CGI-Beispiele zu testen, empfiehlt es sich, diese an Ort und Stelle zu belassen und die Einträge in der Apache-Konfiguration entsprechend anzupassen. Für das erste Beispiel `cgi.basic` bedeutet dies die folgenden Änderungen in der Konfigurationsdatei:

```
#DocumentRoot /var/www/
DocumentRoot <embedded-linux-dir>/examples/cgi/cgi_basic/
...
#ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
ScriptAlias /cgi-bin/ <embedded-linux-dir>/examples/cgi/cgi_basic/cgi-
    bin/
```

<sup>15</sup> Bei installiertem Apache2-Webserver sollte unter Debian die Datei `/etc/init.d/apache2` vorhanden sein. In OpenWrt ist der Webserver `httpd` Teil des Paketes `busybox`.

<sup>16</sup> Je nach Distribution ist dies die Datei `/etc/apache2/httpd.conf` oder `/etc/httpd/httpd.conf`. Unter Debian existiert jedoch eine zusätzliche Datei `/etc/apache2/sites-available/default`, in der diese Einstellungen vorzunehmen sind.

<sup>17</sup> Unter OpenWrt wird das HTML-Wurzelverzeichnis beim Start von `httpd` mit angegeben. Die Einstellungen sind dementsprechend in `/etc/init.d/httpd` anzupassen. Das CGI-Verzeichnis wird bei OpenWrt immer im HTML-Wurzelverzeichnis erwartet.

Die OpenWrt-Beispiele werden am einfachsten auf einem Host-PC mit dem Aufruf `make CROSS=1 cross-compile`, dann wird das gesamte Beispielverzeichnis übertragen.

Nach einem Neustart mittels `sudo /etc/init.d/apache2 restart` sollte unter `http://<embedded-system-ip>` eine Website mit dem Text *click on a button to see the output of this command* und zwei Befehlsknöpfen erscheinen.<sup>18</sup> Ein Mausklick auf eine der Tasten startet das zugehörige Skript und damit den Befehl in der Kommandozeile und liefert die Antwort über den Umweg der Standardausgabe an den Webbrowser zurück. Da in diesem Beispiel keine Daten an die jeweiligen Skripte übertragen werden, ist es irrelevant, ob die Formulardaten mit der *GET*- oder der *POST*-Methode übertragen werden. Die Einbindung der beiden CGI-Skripte in diesem Beispiel entspricht der in Abbildung 13.11 gezeigten Kommunikation.

### 13.4.1 Messdatenanzeige

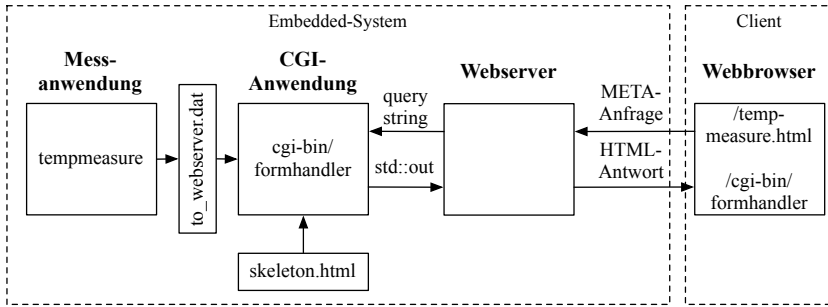
Die CGI-Anwendung wird bei jedem Aufruf in der HTML-Datei gestartet und muss entsprechend unabhängig von einer dauerhaft laufenden Messanwendung programmiert werden. Als Schnittstelle zwischen diesen beiden Anwendungen auf dem Embedded-System kann eine Datei dienen (vgl. Abbildung 13.12). Das Beispiel einer Temperaturmessung `cgi-tempmes` zeigt, wie die Messdaten zyklisch in der Datei `to.webserver.dat` abgelegt werden. Ein separates Programm `formhandler`, das als CGI-Anwendung fungiert, bedient sich aus `to.webserver.dat` und fügt die dort hinterlegten Kombinationen *Bezeichnung-Wert* in das HTML-Skelett `skeleton.html` ein. Eine Anfrage an die CGI-Anwendung wird in diesem Fall nicht über ein Formular gestellt, sondern über folgenden Eintrag im `<meta>`-Tag innerhalb des Kopfbereichs der HTML-Seite:

```
<head>
  <meta http-equiv="refresh" content="1; URL=/cgi-bin/formhandler">
  <title>Temperaturmessung</title>
</head>
```

Mit der Option `refresh` in Kombination mit `content="1;..."` wird die Anfrage automatisch jede Sekunde neu gestellt und die Website damit zyklisch aktualisiert.

Für den Test des Beispiels sind zunächst die Angaben für das Wurzel- und das CGI-Verzeichnis in der Webserver-Konfiguration wie im vorigen Abschnitt

<sup>18</sup> An dieser Stelle sei darauf hingewiesen, dass beim intensiven Test während der Entwicklung eigener Seiten häufiger der Cache des Browsers geleert werden sollte. Nur so können die Änderungen unmittelbar getestet werden. Für den Browser *Firefox* existiert die Erweiterung *Web Developer Toolbar*, um den Cache dauerhaft zu deaktivieren.



**Abb. 13.12.** Unidirektionale Kommunikation zwischen Mess- und CGI-Anwendung im Beispiel `cgi.tempmes`. In der CGI-Anwendung werden Prozessdaten in das Skelett einer HTML-Seite eingebettet. Bei einer Anfrage des Webbrowsers erfolgt die vollständige Übertragung der neuen Seite.

beschrieben zu ändern. Nach einem Neustart des Webserver wäre die Website unter `http://<embedded-system-ip>/tempmeasure.html` erreichbar. Allerdings existiert noch keine Anwendung `formhandler`, die eine umgehend gestellte Anfrage beantworten könnte. Vor dem Übersetzen der beiden Anwendungen `tempmeasure` und `formhandler`<sup>19</sup> ist darauf zu achten, dass sowohl in `Makefile` als auch in `tempmeasure.cpp` die optionalen Angaben `USE_IIC` abgeschaltet sind. Ohne diese Option wird ein virtueller Temperaturwert erzeugt, sodass ein Test auch ohne real angeschlossenen I<sup>2</sup>C-Sensor möglich ist.

Nach dem Übersetzen sollte sich nun beim Aufruf der obigen URL eine Website zeigen, die von der CGI-Anwendung `formhandler` erzeugt wurde. Die Hauptaufgabe von `formhandler` ist es, die in `/dev/shm/to_webserver.dat`<sup>20</sup> hinterlegten Einträge der Form `CGINAME_<NAME> <WERT>` in die entsprechenden Platzhalter in `skeleton.html` einzusetzen. Noch sind keine Messdaten vorhanden. Diese werden durch den Start der Anwendung erzeugt:

```
$ sudo ./tempmeasure
```

Nun sollten Messdaten erscheinen, die jede Sekunde neu aktualisiert werden. Root-Ausführungsrechte sind notwendig, um die Prozess-ID von `tempmeasure` an der in `cgi-bin/config.dat` definierten Stelle ablegen zu können. Vom Programm `formhandler` wird diese Information abgeholt und im System nachgeschaut, ob aktuell ein Prozess mit der dort hinterlegten ID läuft. Dies ist der Fall, wenn die Datei `/proc/<process-id>/status` existiert.

<sup>19</sup> Durch das gemeinsame `Makefile` im Verzeichnis `cgi.tempmes`.

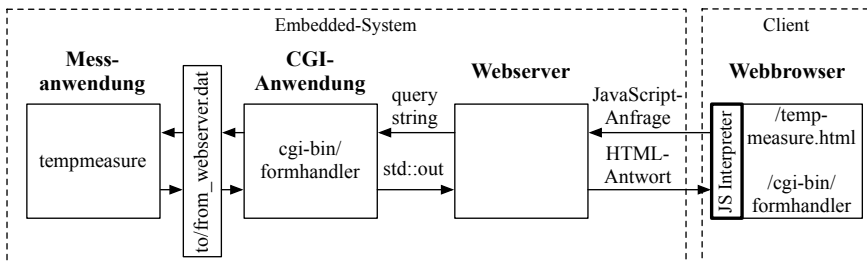
<sup>20</sup> Im Verzeichnis `/dev/shm` ist ein *Shared-Memory*-Bereich gemountet. Das bedeutet, dass sich dort liegende Daten im RAM-Speicher befinden, auf den alle Prozesse Zugriff haben. Schreib- und Lesevorgänge sind in diesem Verzeichnis wesentlich schneller, als in einem Festspeicher-Verzeichnis.

Für einen einfachen Zugriff auf den Inhalt sowohl der Parameterdatei `cgi-bin/config.dat` als auch der Daten `/dev/shm/to.webserver.dat` steht eine Klasse `ParamFile` zur Verfügung.<sup>21</sup> Damit können Einträge der Form `<bezeichner> <wert>` in Dateien abgelegt bzw. aus diesen eingelesen werden. Das Beispiel `param_file_parser` zeigt die Verwendung dieser Klasse.

Steht der in Abschnitt 9.4 vorgestellte Sensor zur Verfügung, so kann das Beispiel durch das Setzen der `USE_IIC`-Platzhalter in Quelltextdatei und Makefile mit echten Werten getestet werden. Details zur Verwendung des I<sup>2</sup>C-Busses und der angeschlossenen Komponenten sind in den Kapiteln 8 und 9 erläutert.

### 13.4.2 Gezielte Anfragen mit JavaScript

In der Implementierung zum vorigen Abschnitt wurde stets die gesamte Website übertragen. Dies ist oftmals nicht wirklich notwendig und auch unter dem Gesichtspunkt der Performanz keine elegante Lösung. Noch schwerer wiegt weiterhin der Nachteil, dass bei zyklischer Aktualisierung durch den `refresh`-Eintrag im `<meta>`-Tag keine Eingaben in Formularfelder gemacht werden können. Diese würden sofort wieder von der neuen Seite überschrieben. Eine Lösung liegt in der Verwendung von JavaScript. Damit lassen sich gezielt Inhalte der Website aktualisieren, ohne, dass diese komplett neu geschrieben werden müsste. Die Realisierung einer bidirektionalen Verbindung mit zyklischer Aktualisierung der Messdaten und Übertragung der Eingabeereignisse ist damit wesentlich einfacher möglich (vgl. Abbildung 13.13). JavaScript besitzt zudem den Vorteil, dass der Java-Quellcode im Browserfenster ausgeführt wird und dadurch nicht das Embedded-System, sondern nur den angeschlossenen Host-Rechner Zeit kostet.



**Abb. 13.13.** Bidirektionale Kommunikation zwischen Mess- und CGI-Anwendung im Beispiel `cgi_tempmes_js`. Die Anforderung einzelner Seiteninhalte geschieht über JavaScript-Funktionen.

Der Formular-Handler muss entsprechend geändert werden, um eintreffende Daten des Webserverns in `/dev/shm/from.server.dat` zu speichern und ein-

<sup>21</sup> Zu finden unter `<embedded-linux-dir>/src/tools/`.

zelse Parameteranfragen oder übermittelte Daten zu übernehmen. In beiden Fällen besitzt der eingehende *Query String* folgende Form:

```
Content-Type: application/x-www-form-urlencoded
CGINAME_DIR=<from,to>&CGINAME_<name>=<value>&...
```

Auf Client-Seite werden die JavaScript-Funktionen `setValues()` und `getValues()` verwendet, um über CGI-Nachrichten mit der *POST*-Methode neue Grenztemperaturen zu setzen bzw. um Sensorwerte anzufordern. Das Wertepaar `CGINAME_DIR=<to,from>` zu Beginn jedes *Query Strings* signalisiert dem Form-Handler die Kommunikationsrichtung. In der JavaScript-Funktion `processContents()` werden eingehende Daten verarbeitet und die einzelnen Elemente der HTML-Seite aktualisiert. Auf die JavaScript-Syntax soll an dieser Stelle nicht näher eingegangen werden. Hierzu sei auf das Online-Buch von Christian Wenz<sup>22</sup> verwiesen.

Um Probleme wegen unterschiedlicher Zugriffsrechte<sup>23</sup> auf die *Shared-Memory*-Dateien zu vermeiden, empfiehlt es sich, diese zunächst anzulegen und mit Lese- und Schreibrechten für alle Benutzer zu versehen:

```
$ touch /dev/shm/to_webserver.dat /dev/shm/from_webserver.dat
$ chmod a+rw /dev/shm/*webserver.dat
```

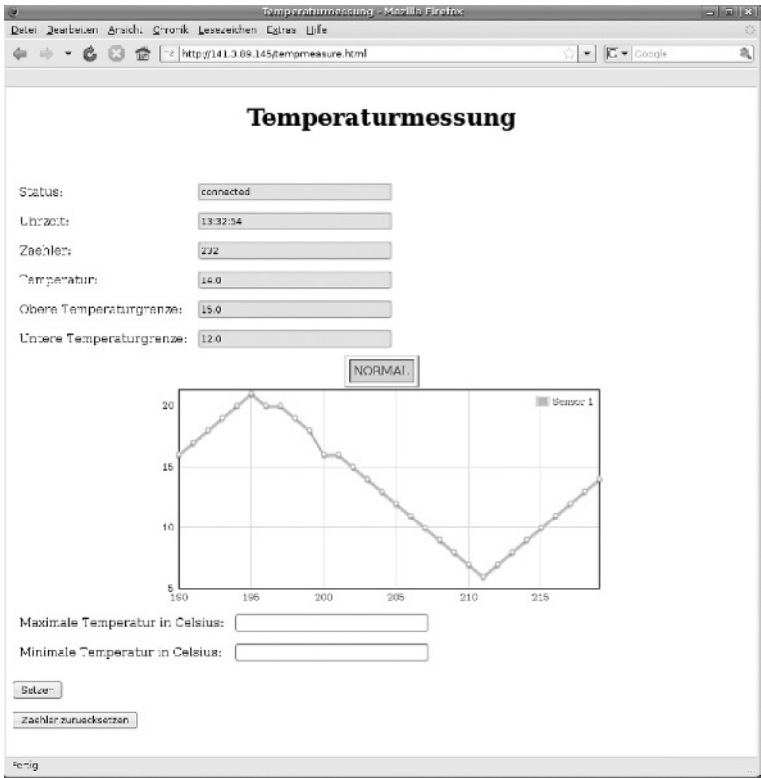
Nach dem Übersetzen der Anwendungen und einem Start von `tempmeasure` werden die Messwerte auf der HTML-Seite zyklisch aktualisiert. Weiterhin lassen sich Eingaben vornehmen, um den Zählstand zurückzusetzen oder Grenztemperaturen zu setzen. JavaScript bietet neben dieser Funktionalität noch wesentlich mehr Möglichkeiten für Erweiterungen: Im Beispiel `cgi.tempmes.plot` wird der Temperaturverlauf mithilfe der JavaScript-Bibliothek *Flot*<sup>24</sup> in einem Diagramm dargestellt (vgl. Abbildung 13.14).

<sup>22</sup> Vgl. [http://openbook.galileocomputing.de/javascript\\_ajax/](http://openbook.galileocomputing.de/javascript_ajax/).

<sup>23</sup> `tempmeasure` wird als Root ausgeführt, `formhandler` als Benutzer `www-data`.

<sup>24</sup> *Flot* ist ein Plugin für das *jQuery*-Framework, vgl. <http://www.ajaxschmiede.de/jquery/flot-diagramme-via-javascript-erstellen/>.





**Abb. 13.14.** Darstellung der Messwerte im Diagramm (vgl. das Beispiel `cgi_tempmes_plot`). Der Plot wurde mit der JavaScript-Bibliothek *Flot* erstellt.

## Video for Linux

### 14.1 Einführung

Im nachfolgenden Text wird die Verwendung einer Kamera unter Linux beschrieben. Die Vorgehensweise für eingebettete Systeme ist jener für Desktop-Systeme ähnlich. Ein Unterschied ist allerdings, dass auf den eingebetteten Systemen viele Tools und Treiber nicht standardmäßig vorhanden sind und nachinstalliert werden müssen.<sup>1</sup> Der Installationsablauf wird exemplarisch am Beispiel des MicroClient Sr. unter Puppy Linux 3.01 beschrieben, lässt sich aber auch auf andere Hardware und Linux-Derivate übertragen. Eine kurze Beschreibung für die NSLU2 unter Debian findet sich am Ende des Abschnitts.

### 14.2 Treiberinstallation und Inbetriebnahme

#### *Puppy Linux auf MicroClient*

Zuerst sollte – wenn es die Ressourcen zulassen – ein Tool wie MPlayer, xawtv, videodog oder camstream installiert werden, um den Bildeinzug kontrollieren zu können. Für das vorliegende System ist der MPlayer vorcompiliert unter <http://dotpups.de/dotpups/Multimedia> als mplayer108.pup erhältlich.

Nach erfolgreicher Installation kann nun auch die USB-Webcam eingesteckt und per lsusb die Gerätebezeichnung der Kamera ermittelt werden. Falls wie im Falle von Puppy Linux das Tool lsusb nicht verfügbar ist, so kann es nachinstalliert werden. Für Puppy Linux ist das Tool unter der folgenden URL vorcompiliert verfügbar: <http://www.moyo.me.uk/pupplinux/download1.htm>.

---

<sup>1</sup> Hierzu und zur nachfolgenden Vorgehensweise vgl. auch <http://tldp.org/HOWTO/Webcam-HOWTO/index.html> und <http://www.murga-linux.com/puppy/viewtopic.php?t=21464>.

Das Entpacken erfolgt mittels:

```
$ gzip -d puppy-lsusb-0.72.tar.gz
$ tar xvf puppy-lsusb-0.72.tar
```

Was für lsusb nun u. U. noch fehlt, ist die aktuelle *Working Devices List*, eine Datei namens **usb.ids**, welche sämtliche bekannte IDs aufführt und Geräten zuordnet. Die Datei ist zu beziehen unter <http://www.linux-usb.org/usb.ids> und zu speichern unter `/usr/local/share/usb.ids`. Ist das Tool **lsusb** mitsamt Liste nun erfolgreich installiert, so kann es folgendermaßen aufgerufen werden:

```
$ lsusb | grep -i video
```

Dies liefert im vorliegenden Beispiel folgende Ausgabe:

```
$ /root/downloads/root/my-applications/bin/lsusb
Bus 003 Device 001: ID 0000:0000
Bus 002 Device 002: ID 0c45:613b Microdia Win2 PC Camera
Bus 002 Device 001: ID 0000:0000
Bus 001 Device 003: ID 046a:0023 Cherry GmbH Cymotion Master Linux Keyboard
Bus 001 Device 002: ID 15ca:00c3 Textech International Ltd. Mini Optical
      Mouse
Bus 001 Device 001: ID 0000:0000
```

Nun kann der MPlayer gestartet werden, um in einem ersten Versuch zu klären, ob die Treiberunterstützung für dieses Gerät bereits vorhanden ist:

```
$ mplayer tv:// -tv driver=v4l:width=640:height=480:device=/dev/video0
```

Im vorliegenden Falle von Puppy 3.01 fehlt allerdings die Unterstützung, der Treiber muss nachinstalliert werden. Nachdem mit **0c45:613b** die Geräte-ID und im Klartext auch der Gerätenamen der Kamera zur Verfügung stehen, kann durch eine Google-Suche rasch die notwendige Treiberversion recherchiert werden. Bei unseren Experimenten hat der **gspca**-Treiber<sup>2</sup> alle unsere vier unterschiedlichen Webcams unterstützt. Er ist als **gspcav1-20070508-i586.pet** unter folgender URL vorcompiliert für Puppy erhältlich:

<http://www.murga-linux.com/puppy/viewtopic.php?mode=attach&id=5467>

Im Quelltext ist der Treiber erhältlich via <http://mxhaard.free.fr>. Hier finden sich auch umfangreiche Listen zur unterstützten Kamera-Hardware. Nach der Installation sollte die Datei **gspca.ko** in folgendem Verzeichnis zu finden sein:

```
/lib/modules/2.6.21.5/kernel/drivers/usb/media
```

Bei abweichender Version oder bei Misslingen kann die Datei aber auch händisch in das u. U. vorher noch anzulegende Verzeichnis `.././media` kopiert werden. Je nach Systemversion bzw. bei abweichenden Verzeichnisnamen muss die Datei noch verschoben werden. In unserem Falle vom

<sup>2</sup> Generic Software Package for Camera Adapters.

Verzeichnis `../2.6.21.5/kernel/drivers/usb/media` in das Verzeichnis `../2.6.21.7/kernel/drivers/usb/media`.

Zur weiteren Installation sind folgende Eingaben notwendig (Für einige der folgenden Befehle sind Root-Rechte erforderlich, hier ist entsprechend `sudo` voranzustellen):

```
$ depmod -ae
$ modprobe videodev
$ modprobe gspca
$ dmesg | grep spca
```

Hierbei kontrolliert der Befehl `depmod -ae` als Vorbereitung die Abhängigkeiten und `modprobe` fügt das Modul ein. Mittels `dmesg | grep spca` lässt sich danach überprüfen, ob das Modul eingebunden ist.

Nach erfolgreicher Installation sollte nun mit dem bereits installierten MPlayer ein Bildeinzug zur ersten Kontrolle möglich sein (zur Fehlerbehandlung vgl. den Text am Ende dieses Abschnitts). Der Aufruf erfolgt, wie oben bereits beschrieben. Alternativ kann an Stelle des MPlayers auch die Version mit grafischer Benutzeroberfläche aufgerufen werden; der Aufruf lautet dann `gmplayer <...>`. Hier können Farben, Sättigung usw. mit Schiebereglern eingestellt werden. Wie schon angesprochen, existieren noch viele andere hilfreiche Tools für erste Versuche mit der Bildquelle: `xawtv`, `videodog`, `camstream`, `caminfo`, `webcamd`. Exemplarisch sei hier das schlanke Tool `videodog` vorgestellt. `Videodog` wird folgendermaßen installiert und übersetzt:

```
$ cd
$ mkdir downloads
$ cd downloads
$ wget http://www.sourceforge.org/Multimedia/Video/Webcam/videodog0.31.tar.gz
$ tar xvf videodog0.31.tar.gz
$ cd videodog-0.31
$ make
$ make install
```

Mit folgenden Aufrufparametern nimmt `videodog` ein Einzelbild von der Webcam auf und legt es als Datei `test.jpg` im JPEG-Format ab:

```
$ ./videodog -x 352 -y 288 -w 3 -d /dev/video0 -j -f test.jpg
```

Da die Tools fast durchgängig unter der GNU-Lizenz im Quelltext vorliegen, kann man sich hier auch das eine oder andere Implementierungsdetail anschauen.

Eine Anmerkung zur Bildqualität: Es ist möglich, sogar relativ wahrscheinlich, dass die Webcam unter Linux bei Weitem nicht die Bildqualität liefert, wie unter Windows. Hierfür gibt es mehrere Gründe: Zum einen sind den Programmierern der freien Treiber nicht alle Details der verwendeten Hardware bekannt. Eine gezielte Optimierung des Bildes mithilfe von Lookup-Tabellen o. ä. entfällt entsprechend. Zum anderen setzen viele Webcams vor oder in der

Linse kombinierte Infrarot- und Ultraviolett-Sperrfilter ein. Da deren Charakteristika nicht bekannt sind, kann das Bild entsprechend auch nicht im Treiber angepasst bzw. rückgerechnet werden. Und abschließend werden bei diesen generischen, bzw. allgemein gehaltenen Treibern auch nicht alle spezifischen Möglichkeiten der Kamera wie Weißabgleich, Gain, Sharpness usw. dem Anwender als einstellbar durchgereicht.

Eine Anmerkung zu auftretenden Treiberkollisionen: Bei unserer Installation des GSPCA-Treibers trat eine Kollision mit zwei vorhandenen Treibern auf. Es handelt sich um den vorinstallierten sn9c102-Treiber und den Video-for-Linux-2-Treiber (V4L2). Die zwei Treiber müssen entsprechend auf die Blacklist gesetzt werden. Unter Puppy geschieht dies mittels Menu / System / Bootmanager / Blacklist. Weiterhin wurde der GSPCA-Treiber nicht automatisch beim Booten geladen. Abhilfe schafft hier ein Eintrag in der Datei `/root/.xinitrc`. Der Eintrag muss vor den letzten Zeilen zum Start des Window-Managers erfolgen und lautet `modprobe gspca`. Die Start-Datei sieht damit aus wie folgt:

```
(...)  
  
modprobe gspca  
#exec $CURRENTWM  
#v2.11 GuestToo suggested this improvement...  
which $CURRENTWM && exec $CURRENTWM  
[ -x $CURRENTWM ] && exec $CURRENTWM  
exec jwm  
###END###
```

Bei neueren Linux-Versionen ist auf jeden Fall ein Test lohnenswert, ob die Kamera nicht bereits von Haus aus unterstützt wird. So ist z.B. bei Puppy Linux Version 4.1 ein verbesserter Webcam-Support angekündigt. Für das vorliegende Tutorial wurde allerdings noch das stabilere (?) Puppy Linux V 3.01 verwendet.

### *Debian auf NSLU2*

Wie bereits angesprochen, wird nun noch kurz der etwas abweichende Ablauf für die NSLU2 unter Debian vorgestellt. Vorausgesetzt wird hierbei ein gem. Kapitel 4 vorbereitetes System. Die erforderliche `gspca`-Erweiterung muss für die NSLU2 als Quellen installiert und dann übersetzt werden:

```
apt-get update  
apt-get install gspca-source  
m-a prepare  
m-a a-i gspca  
modprobe -v gspca  
dmesg | grep spca
```

Da die NSLU2 keine VGA-Schnittstelle besitzt, fehlt die grafische Ausgabe. Für einen ersten Test kann aber, wie schon oben beschrieben, das schlanke

Grabber-Tool videodog nachinstalliert, compiliert und gestartet werden. Abweichend zur Puppy-Einrichtung fehlt hier allerdings noch die Bibliothek mit den jpeg-Routinen:

```
$ apt-get install libjpeg-dev
$ cd
$ mkdir downloads
$ cd downloads
$ wget http://www.sourceforge.org/Multimedia/Video/Webcam/videodog0.31.tar.gz
$ tar xvf videodog0.31.tar.gz
$ cd videodog-0.31
$ make
$ make install
```

Ein Aufruf von videodog zieht nun von der Webcam ein Bild ein und legt es als jpeg-Datei auf dem USB-Stick ab:

```
$ ./videodog -x 352 -y 288 -w 3 -d /dev/video0 -j -f test.jpg
```

Die Datei kann für einen ersten Test per scp übertragen und auf dem Host-PC dargestellt werden.

## 14.3 Bildeinzug unter Linux per V4L

Video-for-Linux oder kurz V4L wurde im Rahmen der Entwicklung des Linux-Kernels 2.1.x eingeführt und ist eine API<sup>3</sup> zum Videobildeinzug unter Linux. Das Interface unterstützt mittlerweile viele (USB-)Webcams, TV-Karten, analoge Framegrabber-Karten und auch Firewire-1394-Kameras. Benannt wurde das Interface in Anlehnung an Video for Windows (VfW), hat aber technisch mit diesem Standard nichts gemein. Mittlerweile, ab Kernel 2.5.x, steht V4L in der zweiten Version als V4L2 zur Verfügung. Zwar bietet V4L2 einen Kompatibilitätsmodus für V4L, hier ist aber Vorsicht geboten: Die Unterstützung ist teilweise nur unvollständig und es wird empfohlen, V4L2-Geräte auch im V4L2-Modus anzusprechen. Zu weiteren Details finden sich untenstehend die wichtigsten Ressourcen zu V4L und V4L2:

- [http://www.linuxtv.org/v4lwiki/index.php/Main\\_Page](http://www.linuxtv.org/v4lwiki/index.php/Main_Page)
- [http://pages.cpsc.ucalgary.ca/~sayles/VFL\\_HowTo](http://pages.cpsc.ucalgary.ca/~sayles/VFL_HowTo)
- <http://v4l.videotechnology.com/dwg/v4l2.pdf>
- <http://www.exploits.org/v4l>
- <http://linux.bytesex.org/v4l2>

Im weiteren Text wird exemplarisch eine einfache Implementierung für einen Bildeinzug von einer Webcam vorgestellt. Die Implementierung basiert auf

<sup>3</sup> Application Programming Interface.

V4L, da dieser Standard eine größere Auswahl an Kameras unterstützt, als V4L2 (vgl. auch [Gräfe 05]). Zur Implementierung vgl. auch das oben genannte, sehr empfehlenswerte Tutorial von Maxwell Sayles, aus welchem hierfür weite Teile übernommen wurden.<sup>4</sup> Für die folgende Implementierung wird eine angeschlossene und funktionsfähige Webcam vorausgesetzt. Die korrekte Funktionsweise kann mit einem Tool wie beispielsweise MPlayer kontrolliert werden, wie in Abschnitt 14.2 erläutert. Weiterhin wird das Vorhandensein des GCC-Compilers auf dem Target bzw. einer (Cross)-Compiler-Toolchain auf dem Entwicklungs-PC vorausgesetzt. Zur Installation dieser Tools vgl. Kapitel 3 bis 6.

Wenn auf dem System eine funktionsfähige V4L-Installation vorhanden ist, so sollte auch die Header-Datei `videodev.h` im Include-Pfad `/usr/include/linux` oder `/usr/src/linux` zu finden sein. Weiterhin ist dann im Regelfall auch eine Hilfe zur API verfügbar unter `/usr/src/linux/Documentation/video4linux/API.html`. Ist die Hilfedatei nicht vorhanden, so ist sie alternativ auch leicht im Web zu finden.

Ein Minimalbeispiel mit Namen `example.cc` ist unter der URL von Maxwell Sayles zu finden. Hiermit wird das Device geöffnet, die Eigenschaften werden abgefragt bzw. eingestellt, dann werden zehn Bilder eingelesen und im PPM-Format<sup>5</sup> auf der Platte abgelegt. Die einzelnen notwendigen Schritte hierzu sind in den folgenden Listing-Blöcken erläutert (zu den erforderlichen Variablen bzw. Deklarationen vgl. auch `videodev.h`).

Für die vorliegende V4L-Implementierung<sup>6</sup> sind die folgenden Includes Voraussetzung:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <linux/videodev.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
```

Im ersten Schritt wird das Video-Device geöffnet und bei Erfolg ein Handle darauf zurückgegeben:

```
int deviceHandle;
char* deviceName = "/dev/video";
```

<sup>4</sup> An dieser Stelle nochmals vielen Dank an Maxwell Sayles für die Erlaubnis, die Inhalte zu verwenden.

<sup>5</sup> Das sog. Portable-Pixmap-Format stellt das einfachste denkbare Bilddateiformat dar (vgl. auch den letzten Listing-Block bzw. [Wikipedia 08, Portable.Pixmap]). Die Dateien können unter Linux mit den gängigen Viewern betrachtet werden.

<sup>6</sup> Die Quellen zum vorliegenden Kapitel finden sich im Unterverzeichnis `<embedded-linux-dir>/src/video4linux`.

```
deviceHandle = open (deviceName, O_RDWR);
if (deviceHandle == -1)
{ // Fehler beim Oeffnen des Device, return -1 ...
}
```

Der Name des Devices lautet standardmäßig `/dev/video` oder `/dev/video0`. Im nächsten Schritt werden die Eigenschaften des Gerätes abgefragt. Der Datenaustausch mit dem Gerät erfolgt über die `ioctl()`-Funktion (IO-Control), die einen Datenaustausch von Programmen im User Space mit Programmen im Kernel Mode bzw. Gerätetreibern realisiert. Zum mitgegebenen Struct vgl. die Header-Datei `videodev.h` und die anfangs erwähnte Hilfedatei.

```
struct video_capability capability;
if (ioctl(deviceHandle, VIDIOCGCAP, &capability) != -1)
{ // Anfrage erfolgreich
}
else
{ // Anfrage gescheitert, return -1 ...
}

if ((capability.type & VID_TYPE_CAPTURE) != 0)
{ // Erfolgreich, das Geraet kann Videodaten im Speicher ablegen
}
else
{ // Anfrage gescheitert, return -1 ...
}
```

Im nächsten Schritt werden die Videokanäle des Gerätes durchnummeriert. Ein Beispiel für ein Gerät mit mehreren Kanälen ist eine TV-Karte, die auch einen S-Video- bzw. AV-Eingang besitzt. Der Schritt ist optional, wenn das Gerät wie bspw. unsere Webcam nur einen Kanal besitzt, oder wenn die Nummer des Kanals bekannt ist. Ansonsten sollte dem Anwender die Möglichkeit geboten werden, den Kanal auszuwählen. Anschließend wird der gewählte Kanal gesetzt. Weiterhin kann optional die Variable `selectedChannel.norm` gesetzt werden zu `VIDEO_MODE_PAL`, `VIDEO_MODE_SECAM` oder `VIDEO_MODE_AUTO`.

Wenn das Video-Gerät die Funktion der Bildskalierung unterstützt, können weiterhin Breite und Höhe des Bildes eingestellt werden.

```
struct video_channel queryChannel;
i = 0;
while (i < capability.channels)
{
    queryChannel.channel = i;
    if (ioctl (deviceHandle, VIDIOCGCHAN, &queryChannel) != -1)
    { // Erfolgreich. Informationen im struct querychannel.xx
        printf ("%d. %s\n", queryChannel.channel, queryChannel.name);
    }
    else
    { // Anfrage gescheitert,
      // kein schwerwiegender Fehler ...
    }
    ++i;
}

printf ("Select a channel:\n");
fflush (stdout);
scanf ("%d", &channelNumber);
```



```

selectedChannel.channel = channelNumber;
if (ioctl (deviceHandle, VIDIOCCHAN, &selectedChannel) == -1)
{ // Setzen des Kanals gescheitert
    printf ("Kanal #%d\n konnte nicht gesetzt werden.", channelNumber);
}

// Hier fehlt: Eingabe von width, height durch den Anwender
// oder Vorgabe (Bsp.: 640, 480)

if ((capability.type & VID_TYPE_SCALES) != 0)
{ // Geraet unterstuetzt Skalierung
    struct video_window captureWindow;

    captureWindow.x = 0;
    captureWindow.y = 0;
    captureWindow.width = width;
    captureWindow.height = height;
    captureWindow.chromaKey = 0;
    captureWindow.flags = 0;
    captureWindow.clips = 0;
    captureWindow.clipcount = 0;
    if (ioctl (deviceHandle, VIDIOCSWIN, &captureWindow) == -1)
    { // Setzen der neuen Werte gescheitert,
      // kein schwerwiegender Fehler ...
    }
}
}

```

Die Abfrage der aktuell eingestellten Breite und Höhe geschieht folgendermaßen:

```

if (ioctl (deviceHandle, VIDIOCGWIN, &captureWindow) == -1)
{ // Anfrage gescheitert
}
width = captureWindow.width;
height = captureWindow.height;

```

Genauso können auch Farbtiefe und Farbpalette eingestellt bzw. abgefragt werden. Wenn mit den Vorgabewerten gearbeitet wird, sind diese Funktionen optional.

```

struct video_picture imageProperties;

if (ioctl (deviceHandle, VIDIOCGPICT, &imageProperties) != -1)
{ // Abfrage erfolgreich
    // Die folgenden Werte stellen den 8-Bit-Graustufenmodus ein:
    imageProperties.depth = 8;
    imageProperties.palette = VIDEO_PALETTE_GREY;
    if (ioctl (deviceHandle, VIDIOCSPICT, &imageProperties) == -1)
    { // Setzen der neuen Werte gescheitert,
      // kein schwerwiegender Fehler ...
    }
}

int depth;
int palette;
struct video_picture imageProperties;

if (ioctl (deviceHandle, VIDIOCGPICT, &imageProperties) == -1)
{ // Abfrage der aktuellen Werte für den Bildmodus gescheitert,
  // kein schwerwiegender Fehler ...
}
}

```

```
depth = imageProperties.depth;
palette = imageProperties.palette;

// wenn im aufrufenden Programm RGB24 erforderlich ist,
// so sollte dieser Modus hier abgefragt werden:

if ((depth != 24) || (palette != VIDEO_PALETTE_RGB24))
{ // Format wird nicht unterstuetzt
}
```

Wenn die Eigenschaften des Gerätes bekannt sind und alle notwendigen Einstellungen vorgenommen wurden, so kann ein Bildeinzug erfolgen. Die einfachste Möglichkeit ist ein Lesezugriff auf das Device. Dieser (blockierende) Lesevorgang wartet so lange, bis ein vollständiges Bild im Speicher ist. Ein Bildeinzug über diesen Mechanismus wird nicht von allen Geräten unterstützt und ist generell nicht zu empfehlen.

Die zweite, empfehlenswertere Möglichkeit ist ein Zugriff über Memory Mapped Input/Output (MMIO). Hierbei wird der Hardware-Speicher des Videogerätes direkt in das RAM des Rechners abgebildet (gemappt) und kann dort über Pointer ausgelesen werden. Hierfür werden zuerst vom Device die Kenngrößen des Bildspeichers erfragt. Dies sind: Bildgröße in Bytes, Anzahl gepufferter Frames und ein Offset-Array für einen Zugriff auf die Frames im Bildspeicher. Im nächsten Schritt wird ein Pointer auf den Speicherbereich gesetzt.

```
struct video_mbuf memoryBuffer;
if (ioctl (deviceHandle, VIDIOCMBUF, &memoryBuffer) == -1)
{ // Abfrage gescheitert, return -1 ...
}

char* memoryMap;
memoryMap = (char*)mmap (0, memoryBuffer.size, PROT_READ | PROT_WRITE,
    MAP_SHARED, deviceHandle, 0);
if ((int)memoryMap == -1)
{ // Operation gescheitert, return -1 ...
}
```

Die Adressen der einzelnen Bilder im Bildspeicher kann nun über den char-Pointer in Verbindung mit den Daten aus dem Offset-Array errechnet werden:

Bild 0 liegt an Adresse (memoryMap + memoryBuffer.offsets[0])

Bild 1 liegt an Adresse (memoryMap + memoryBuffer.offsets[1])

(...)

Hierbei wird die Anzahl der abgelegten Bilder in der Variablen `memoryBuffer.frames` gespeichert.

Der anschließende Bildeinzug muss über eine Variable vom Typ `video_mmap` erfolgen. Diese Struct-Variable muss angelegt, und die einzelnen Parameterfelder müssen vorbelegt bzw. erfragt werden:

```
struct video_mmap* mmap;
mmap = (struct video_mmap*)(malloc (memoryBuffer.frames * sizeof (struct
    video_mmap)));
```

```

int i = 0;
while (i < memoryBuffer.frames){
    mmaps[i].frame = i;
    mmaps[i].width = width;
    mmaps[i].height = height;
    mmaps[i].format = palette;
    ++i;
}

```

Im weiteren Quelltext wird nun ein Ringpuffer realisiert, und es wird jedes Mal ein Capture Request ausgeführt. Weiterhin wird ein Index für den aktuellen Buffer mitgeführt. Diese Implementierung lässt sich in folgender Funktion `NextFrame()` zusammenfassen, die die Adresse des aktuellen Frame Buffers zurückgibt:

```

int bufferIndex;
bufferIndex = memoryBuffer.frames-1;

char* NextFrame()
{
    // Capture Request auf den aktuellen Buffer
    if (ioctl (deviceHandle, VIDIOCMCAPTURE, &mmaps[bufferIndex]) == -1)
    { // Capture Request gescheitert ...
    }
    // bufferIndex auf den nächsten Frame
    ++bufferIndex;
    if (bufferIndex == memoryBuffer.frames)
    { // bufferIndex zeigt hinter den letzten Buffer
        // -> bufferIndex auf den ersten Buffer
        bufferIndex = 0;
    }
    // Warten, bis Capture-Vorgang auf aktuellen Frame abgeschlossen
    if (ioctl (deviceHandle, VIDIOCSYNC, &mmaps[bufferIndex]) == -1)
    { // Sync-Request gescheitert ...
    }
    // Rueckgabe der Adresse zu den Daten des aktuellen Frames,
    // entsprechend dem aktuellen bufferIndex
    return (memoryMap + memoryBuffer.offsets[bufferIndex]);
}

```

Interessant ist hierbei, dass der Hardware-Capture-Vorgang und die Verarbeitung der Daten parallel, aber auf unterschiedliche Buffer erfolgen. Vgl. folgenden Pseudo-Code:

```

VIDIOCMCAPTURE(0)

while (!hell_freezes_over)
{
    VIDIOCMCAPTURE(1)
    VIDIOCSYNC(0)
    // Verarbeite Frame 0 waehrend die Hardware Frame 1 aufnimmt
    VIDIOCMCAPTURE(0)
    VIDIOCSYNC(1)
    // Verarbeite Frame 1 waehrend die Hardware Frame 0 aufnimmt
    // Dieses Vorgehen ist nicht auf zwei Buffer beschraenkt!
}

```

Unter Verwendung der von der Funktion `NextFrame()` zurückgelieferten Adresse ist nun auch die Ausgabe des Bildes als PPM-Datei leicht möglich. Die

entstandene Datei kann mit einem der gängigen Viewer unter Linux dargestellt werden:

```
char* frame = NextFrame();
char fname[80];
sprintf (fname, "output.ppm");
printf ("Writing out PPM file %s\n", fname);
FILE* fp;
if ((fp = fopen (fname, "w")) == NULL)
{
    printf ("Could not open file %s for writing.\n", fname);
    return -1;
}

fprintf (fp, "P6\n%d %d\n255\n", width, height);
int n = width * height;

for (int index = 0; index < n; ++index)
{
    putc (frame[index*3+2], fp);
    putc (frame[index*3+1], fp);
    putc (frame[index*3+0], fp);
}

fflush (fp);
fclose (fp);

printf ("Use 'xv output.ppm' to view output.\n");
```

Die Implementierung schließt mit einigen Aufräumarbeiten:

```
free (mmaps);
munmap (memoryMap, memoryBuffer.size);
close (deviceHandle);
```

Eine Anmerkung zur Einstellung der Kameraparameter: Auch die Kameraparameter wie Gain, Shutter und Sharpness usw. können über das V4L-Interface eingestellt werden. Wenn hierzu eine der bekannten Videoapplikationen unter Linux verwendet wird, so speichert der Treiber die Einstellungen bis zum Start der eigenen Applikation. Folgende Tools stehen u. a. zur Auswahl: MPlayer, xawtv, videodog, camstream , caminfo, webcamd.

Eine Anmerkung zum V4L2-Standard: Für eine Implementierung eines Bildeinzuges nach dem moderneren V4L2-Standard kann der Quelltext `cvcap_v4l.cpp` der OpenCV-Bibliothek als Basis verwendet werden. Wenn eine OpenCV-Installation auf dem PC vorhanden ist, so befindet sich die Datei im Pfad `../OpenCV/otherlibs/highgui`. Auch für die IVT-Bibliothek ist ein Bildeinzug nach V4L2 geplant; hier lohnt es sich entsprechend, die IVT-Seite bei sourceforge zu beobachten: <http://ivt.sourceforge.net>. Aktuell ist in der IVT dieser Standard über die OpenCV-Schnittstelle angebunden (Dateien `OpenCVCapture.cpp` und `OpenCVCapture.h`).

## 14.4 Treiberkapselung für die IVT-Bibliothek

Die IVT-Bibliothek bietet grundsätzlich bereits eine Video-for-Linux-Schnittstelle, allerdings nur über den Umweg des Anschlusses der OpenCV-Bibliothek (zu Details vgl. Anhang C). Da die OpenCV zwar für ein Prototyping auf dem Entwicklungs-PC recht brauchbar, für eine Verwendung auf dem eingebetteten System aber normalerweise zu voluminös ist, soll diese Schnittstelle hier nicht verwendet werden. Stattdessen wird der Bildeinzug schlanker mit reiner IVT-Funktionalität realisiert. Die Grundlagen zur Kapselung eines Kameras-treibers oder einer anderen Art des Bildeinzugs zur komfortablen Verwendung mit der IVT-Bibliothek werden in Anhang C.2.3 ausführlich beschrieben. Weiterhin liegen im IVT-Verzeichnis `../IVT/src/VideoCapture` viele Beispielimplementierungen vor.

Nachdem in Abschnitt 14.3 bereits eine Bilddatenaufnahme von einem V4L-Gerät in C/C++ realisiert wurde, müssen nun ausgehend von dieser Implementierung nur noch die einzelnen vorliegenden Funktionen wie `OpenCamera()`, `GetWidth()` usw. in einer Klasse zusammengefasst werden. Diese Klasse muss am Ende eine Schnittstelle gem. Abbildung C.4 aufweisen. Ein guter Ausgangspunkt ist es, als Implementierungsgerüst die Dateien `CMU1394Capture.cpp` und `CMU1394Capture.h` zu verwenden.

Die dieserart abgeleitete Implementierung findet sich in den zwei Dateien `VFLinuxCapture.cpp` und `VFLinuxCapture.h`. Die Anwendung der Kapselung wird im nachfolgenden Listing kurz vorgestellt.<sup>7</sup>

Die vorgestellte Implementierung wurde mit den Kameras gem. Tabelle 14.1 erfolgreich getestet.

Hersteller	Typ	ID
Speed Link	Microdia PC Camera (SN9C120) mit integrierter Beleuchtung	0c45:613c
SilverCrest	Microdia Win2 PC Camera	0c45:613b
Pixart Imaging, Inc.	Model DC-3110, Q-TEC WEBCAM 100	093a:2460
Logitech Inc.	QuickCam Express	046d:0928

**Tabelle 14.1.** Auflistung der verwendeten Kameras.

<sup>7</sup> Das Snippet entstammt [Azad 07, Kap. 3].

```

#include <stdio.h>
#include "Image/ByteImage.h"
#include "gui/QTApplicationHandler.h"
#include "gui/QTWindow.h"
// #include "VideoCapture/OpenCVCapture.h"
#include "VideoCapture/VFLinuxCapture.h"

int main(int argc, char **args)
{
    // Kameramodul anlegen

    // COpenCVCapture capture(-1);
    CVFLinuxCapture capture(0); // Channel=0

    // Kamera oeffnen
    if (!capture.OpenCamera())
    {
        printf("Fehler: Konnte Kamera nicht oeffnen.\n");
        return 1;
    }

    const int width = capture.GetWidth();
    const int height = capture.GetHeight();
    const CByteImage::ImageType type = capture.GetType();

    CByteImage *ppImages[] = { new CByteImage(width, height, type) };

    // QT initialisieren
    CQTApplicationHandler qtApplicationHandler(argc, args);
    qtApplicationHandler.Reset();

    // Fenster anlegen und anzeigen
    CQTWindow window(width, height);
    window.Show();

    // Hauptschleife
    while (!qtApplicationHandler.ProcessEventsAndGetExit())
    {
        if (!capture.CaptureImage(ppImages))
            break;

        window.DrawImage(ppImages[0]);
    }

    delete ppImages[0];
    return 0;
}

```

Wenn unter Linux eine bestimmte Kamera zur Anwendung kommen soll, für die es bislang noch keine Treiberunterstützung gibt, so können die folgenden Quellen helfen, einen eigenen Treiber in Angriff zu nehmen bzw. einen Windows-Treiber nachzubilden (vgl. auch google: "reverse engineering" linux webcam how-to).

<http://mxhaard.free.fr/dijonrml1.pdf>

<http://www.mnementh.co.uk/home/projects/linux/sonix/sonix>

<http://info.i.et.unipi.it/~luigi/FreeBSD/usb-cameras.html>

<http://groups.google.com/group/microdia/web/reverse-engineering-windows-drivers>

## Intelligente Kamera

### 15.1 Einführung

Nachdem in Kapitel 14 bereits der Anschluss einer Kamera an einen Embedded-PC erklärt wurde, wird nun im vorliegenden Kapitel als Beispielapplikation hierfür eine Anwendung aus dem Bereich der Sicherheitstechnik bzw. Bewegungserkennung entwickelt.<sup>1</sup> Dann wird der Umbau einer Webcam auf einen Standard-Objektivanschluss vorgestellt. Abschließend wird die Möglichkeit der Hardware-Triggerung und der Anschluss einer Firewire-Kamera erläutert, und es werden weiterführende Hinweise zu komplexeren Applikationen gegeben.

### 15.2 Sicherheitssystem mit Bewegungserkennung

Überwachungskameras begegnen uns mittlerweile überall im Alltag. Die Auswertung der Videodaten erfolgt im Regelfall durch den Menschen, aber auch eine maschinelle Auswertung ist denkbar. Als Applikation für eine Smart Camera haben wir uns das folgende fiktive Szenario ausgedacht:

Die Museumsverwaltung des Louvre in Paris sucht zur Überwachung der Mona Lisa eine Alternative zu den bisher eingesetzten Laserlichtschranken, da diese zu gefährlich für die Augen der Besucher sind. Es soll nun eine kamerabasierte Lösung eingesetzt werden. Für einen zusätzlichen Desktop-PC ist kein Platz vorhanden, entsprechend muss eine intelligente Kamera zum Einsatz kommen (vgl. Abbildung 15.1). Abschließend stehen dem Museum aufgrund von Budget-Kürzungen nur rund 130 EUR zur Verfügung; der Einsatz eines

---

<sup>1</sup> Die Quellen zum vorliegenden Kapitel finden sich im Unterverzeichnis `<embedded-linux-dir>/src/video4linux`.



kommerziellen, schlüsselfertigen Smart-Camera-Systems aus der Liste am Ende des Anhangs B kommt entsprechend nicht infrage.

Unsere Umsetzung sieht nun aus wie folgt: Als Hardware-Basis wird eine NSLU2 verwendet, an welche eine preiswerte USB-Webcam angeschlossen ist.<sup>2</sup> Aus Kostengründen wird in einer ersten Phase das mitgelieferte Plastikobjektiv der Webcam verwendet. In der nächsten Phase werden die Komponenten aber in ein industrietaugliches Alu-Gehäuse eingebaut, und die Kamera erhält einen C-Mount-Objektivanschluss (Abschnitt 15.3.1). Der Bildeinzug geschieht wieder über das V4L-Interface bzw. über die Kapselung in der IVT-Klasse `CVFLinuxCapture` der IVT-Bibliothek. Die Implementierung des Bewegungserkenners wurde aus [Azad 07, Kap. 4] entnommen und auf den neuen Bildeinzug umgestellt.



**Abb. 15.1.** Überwachung der Mona Lisa im Louvre mit einer Smart Camera.<sup>3</sup>

Die Ausgabe, bzw. das Auslösen des Alarms könnte nun bei der NSLU2 über Netzwerk oder über einen der GPIO-Pins<sup>4</sup> erfolgen. Für ein schnelles Prototyping wird die Applikation aber zuerst auf dem Desktop-PC umgesetzt und

<sup>2</sup> Alle im Folgenden benötigten Teile sind in der Teileliste in Anhang E aufgeführt.

<sup>3</sup> Herzlichen Dank an Alexander Kasper, der diese anschauliche Grafik erstellt hat.

<sup>4</sup> Zweckentfremdete Anschlüsse der Status-LEDs (vgl. Abschnitt 7.4).

evaluiert, und hier erfolgt zu Testzwecken die Ausgabe herkömmlich visuell in einem Qt-Window.

Der Algorithmus zur Bewegungserkennung funktioniert wie folgt: In der aufgenommenen Bildsequenz werden durch eine Differenzbildung die Unterschiede aufeinanderfolgender Bilder erkannt. Mit einer anschließenden Schwellwertsegmentierung werden diese Grauwertdifferenzen extrahiert und binarisiert. Abschließend werden zwei morphologische Operationen angewandt (Dilatation und Erosion), um die Erkennung robuster zu gestalten. Die verwendeten Algorithmen sind in [Azad 07, Kap. 2, 4] detailliert beschrieben.

```
#include "Image/ByteImage.h"
#include "Image/ImageProcessor.h"
#include "Image/PrimitivesDrawer.h"
#include "gui/QTWindow.h"
#include "gui/QTApplicationHandler.h"
// #include "VideoCapture/OpenCVCapture.h"
#include "VideoCapture/VFLinuxCapture.h"
#include <stdio.h>

int main(int argc, char **args)
{
    // Kameramodul anlegen und initialisieren
    // COpenCVCapture capture(-1);
    CVLinuxCapture capture(0); // Channel=0

    if (!capture.OpenCamera())
    {
        printf("error: could not open camera\n");
        return 1;
    }

    const int width = capture.GetWidth();
    const int height = capture.GetHeight();
    const int nPixels = width * height;

    CByteImage *ppImage[]={new CByteImage(width,height,CByteImage::eRGB24)};
    CByteImage grayImage(width, height, CByteImage::eGrayScale);
    CByteImage lastImage(width, height, CByteImage::eGrayScale);
    CByteImage motionImage(width, height, CByteImage::eGrayScale);

    // QT initialisieren
    CQTApplicationHandler qtApplicationHandler(argc, args);
    qtApplicationHandler.Reset();

    // Fenster anlegen und anzeigen
    CQTWindow window(2 * width, height);
    window.Show();

    // Hauptschleife
    while (!qtApplicationHandler.ProcessEventsAndGetExit())
    {
        capture.CaptureImage(ppImage);

        // Kamerabild in Graustufenbild umwandeln (falls notwendig)
        ImageProcessor::ConvertImage(ppImage[0], &grayImage);

        // Differenzbild erstellen
        for (int j = 0; j < nPixels; j++)
            motionImage.pixels[j]=abs(grayImage.pixels[j]-lastImage.pixels[j]);
```

```

// Differenzbild binarisieren
ImageProcessor::ThresholdBinarize(&motionImage, &motionImage, 20);

// Morphologische Operatoren anwenden
ImageProcessor::Erode (&motionImage, &motionImage, 3);
ImageProcessor::Dilate(&motionImage, &motionImage, 33);
ImageProcessor::Dilate(&motionImage, &motionImage, 33);
ImageProcessor::Dilate(&motionImage, &motionImage, 33);
ImageProcessor::Erode (&motionImage, &motionImage, 33);
ImageProcessor::Erode (&motionImage, &motionImage, 33);
ImageProcessor::Erode (&motionImage, &motionImage, 33);

// Zusammenhängende Bereiche erkennen
RegionList resultList;
ImageProcessor::FindRegions(&motionImage, resultList, 100);

// Erkannte Bereiche visualisieren
for (int i = 0; i < (int) resultList.size(); i++)
{
    const MyRegion &region = resultList.at(i);
    PrimitivesDrawer::DrawRegion(ppImage[0], region, 0, 0, 255, 3);
}

// aktuelles Kamerabild abspeichern
ImageProcessor::CopyImage(&grayImage, &lastImage);

// Ergebnis anzeigen
window.DrawImage(ppImage[0]);
window.DrawImage(&motionImage, width, 0);
}

delete ppImage[0];
return 0;
}

```

Nach erfolgreichem Test auf dem Entwicklungs-PC können nun die Qt-Includes und Qt-Aufrufe aus dem Programm entfernt werden. Stattdessen wird dann bspw. das Schalten eines digitalen Ausgangs implementiert, welcher die Alarmanlage schaltet. Dann kann das Programm auf der NSLU2 übersetzt und aktiviert werden.

Anmerkung: Eine interessante und einfach umzusetzende Option bei dieser Anwendung ist auch die Beschränkung des Alarmbereiches auf ein bestimmtes Rechteck im Bild – für den Fall, dass die Mona Lisa nicht bildfüllend erfasst wird und Bewegungen in den Bereichen außerhalb des Bildes keinen Alarm auslösen sollen.

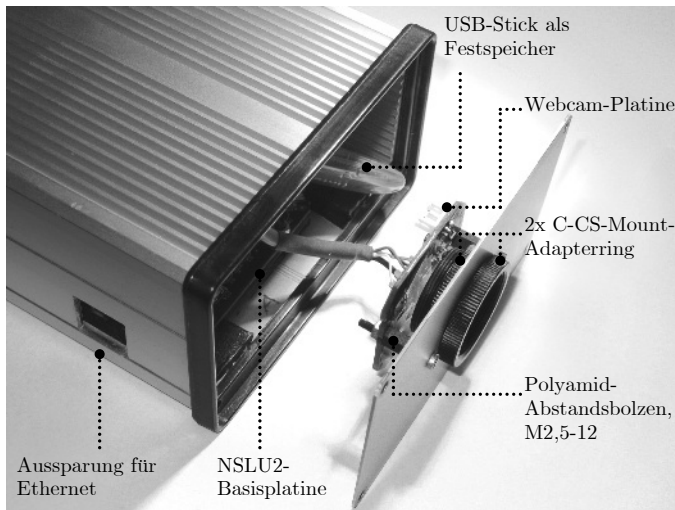
## 15.3 Weiterführende Informationen

### 15.3.1 Kommentare zum Hardware-Aufbau

Bei der vorgestellten Lösung ist der Anwender auf die Brennweite des Webcam-Objektives festgelegt. Weiterhin sind diese kleinen Plastikobjektive lichtschwach und weisen hohe Verzeichnungen auf. Wünschenswert wäre die

Möglichkeit, auch hochwertige Objektive einsetzen zu können. Bei USB- oder Firewire-Kameras, die für den industriellen Einsatz hergestellt werden, ist dies naturgemäß gegeben, die Kameras liegen aber preislich Faktor 10–20 über einer preiswerten Webcam (zu noch vergleichsweise günstigen Exemplaren siehe beispielsweise [TIS 08]).

Unter Kenntnis der notwendigen Geometrie und mit zwei C-CS-Mount-Adapterringen und zwei M2,5-12mm-Abstandsbolzen ist eine Aufnahme für handelsübliche, hochwertige C-Mount-Objektive aber auch relativ leicht selbst zu realisieren (Abbildung 15.2).



**Abb. 15.2.** Umbau einer preiswerten Webcam auf C-Mount-Anschluss. Im Gehäuse ist hier die Basisplatine der NSLU2 eingebaut.

Einige Details zur Vorgehensweise: Der C-Mount-Standard definiert Objektive, die einen Ein-Zoll-Gewindeanschluss mit einer Gewindesteigung von 1/32 Zoll besitzen. Weiterhin beträgt das Auflagemaß, respektive der Abstand zwischen Objektivauflegekante und Sensorfläche, 17,526 mm [Wikipedia 08, C-Mount]. Für die vorliegende Anwendung ist das zöllische Feingewinde kaum selbst herzustellen, es können aber zwei C-CS-Mount-Adapterringe verwendet und – das Frontblech einspannend – ineinandergeschraubt werden.

Als Abstandshalter für die Kameraplatine der zerlegten Webcam kommen zwei Polyamid-Bolzen zum Einsatz. Im Frontblech des Gehäuses muss entsprechend eine Aussparung mit Durchmesser 25 mm für das Objektiv hergestellt, und es müssen zwei 2,5 mm-Löcher für die Bolzen gebohrt werden. Abschließend sollte noch vorsichtig der UV-IR-Sperrfilter aus dem Webcam-Objektiv herausoperiert werden und mit Tesafilm über dem sensitiven Teil des CMOS-Sensors



**Abb. 15.3.** Innenleben der Smart Camera auf Basis der NSLU2.

aufgeklebt werden (der Sperrfilter ist das kleine grünliche Glasplättchen, ca.  $3 \times 3 \text{ mm}^2$ ).

Als Gehäuse haben wir ein Hammond-Alugehäuse vom Distributor Farnell gewählt, in welches die NSLU2-Basisplatine isoliert eingebaut wird (ein Nachmessen von GND gegen Gehäuse empfiehlt sich hier). Die USB-Buchsen auf der Platine mussten gegen stehende Versionen getauscht und die USB-Stecker mit einem Skalpell etwas verschlankt werden, sonst waren außer den Ausschnitten und Bohrungen im Gehäuse keine Umbauten notwendig.

Als Webcam kommt eine sehr preiswerte Logitech Quickcam Express mit einer nativen Auflösung von  $352 \times 288$  zum Einsatz (Abbildung 15.4), bei der das Anschlusskabel etwas gekürzt wurde. Festspeichermedium ist ein preiswerter Intenso-USB-Stick mit zwei Gigabyte, welcher über ein kurzes USB-Kabel angeschlossen ist. Daten und Bezugsquellen zu den Einzelteilen sind in der Tabelle in Anhang E aufgeführt.

### 15.3.2 Triggerung und IEEE 1394-Übertragung

In der industriellen Bildverarbeitung wird oft der Mechanismus eines sog. Hardware-Triggers eingesetzt. Hierbei löst bspw. eine Lichtschranke über dem Förderband, die ein ankommendes, zu prüfendes Teil erfasst, den Bildeinzug der Kamera aus. Webcams aus dem Consumer-Bereich besitzen diese Möglichkeit nicht, und für die Mona-Lisa-Applikation ist das Feature auch nicht erforderlich. Sollte aber für eine andere Anwendung das Feature einmal benötigt werden, so können die genannten (USB)-Kameras der Fir-



**Abb. 15.4.** Fertiggestellte Smart Camera auf Basis der NSLU2.

ma TheImagingSource zum Einsatz kommen [TIS 08]. Eine Open-Source-Implementierung hierzu, die das Feature unterstützt und auch im Quelltext vorliegt, ist:

<http://www.unicap-imaging.org>

Eine detaillierte Anleitung zum Einsatz der Software findet sich unter der folgenden URL:

<http://unicap-imaging.org/blog/index.php?/archives/7-Installing-USB-Cameras.html>

Bei der Verwendung von Unicap ist das Lizenzmodell zu beachten: Unicap wird im sog. Dual Licensing-Modell wahlweise als GNU-Version oder als kommerzielle Version angeboten.<sup>5</sup>

Ein weiteres interessantes Thema ist der Anschluss von IEEE 1394-Firewire-Kameras. Tatsächlich gibt es im Consumer-Bereich nur sehr wenige 1394-Webcams<sup>6</sup>, hier hat sich der USB-Standard durchgesetzt. Im industriellen Bereich wiederum ist der 1394-Übertragungsstandard wesentlich verbreiteter als USB. Es lohnt sich also, die Möglichkeit des Anschlusses einer 1394-Kamera zu kennen. Unter Linux kann der Anschluss über die Bibliotheken libdc1394 und libraw1394 erfolgen – die genaue Vorgehensweise ist im Anhang C beschrieben.

<sup>5</sup> Ansprechpartner ist Arne Caspari: [arne@imaging.org](mailto:arne@imaging.org). An dieser Stelle möchten wir Herrn Caspari auch nochmals herzlich für die Informationen danken.

<sup>6</sup> Apple iSight, Unibrain Fire-i – beide sind nur noch gebraucht erhältlich.

Eine Anmerkung: Von den in Teil 1 vorgestellten Embedded-PCs besitzt nur das Atom-PC-Board der Firma Intel die Möglichkeit, eine 1394-Schnittstelle über PCI nachzurüsten (vgl. Abschnitt 2.6) – hier ist also Vorsicht geboten. U. U. existiert noch die Möglichkeit, ein 1394-Interface für den MicroClient-PC über Mini-PCI nachzurüsten, genauere Erfahrungen liegen uns hierzu aber nicht vor.

### 15.3.3 Weitere Anwendungen

Grundsätzlich kann eine intelligente Kamera für die gleichen Applikationen zum Einsatz kommen, wie ein PC-basiertes System. Der Nachteil der geringeren Rechenleistung wird durch den geringeren Preis, die erhöhte Robustheit und die kleinere Bauform wettgemacht, und so ist aktuell ein starker Trend hin zu Smart Cameras bzw. Vision-Sensoren erkennbar.

Einige interessante Applikationen wie Barcode-Erkennung, Stanzteil-Vermessung, Laserscanning, Objekterkennung und Stereosehen haben wir bereits mitsamt Quelltext im Buch *Computer Vision – Das Praxisbuch* zusammengestellt [Azad 07]. Alle diese Beispiele sind grundsätzlich auch auf eingebettete Systeme übertragbar. Die Quelltextsammlung ist über die Praxisbuch-Einstiegsseite und den dortigen Link zu Computer Vision frei erhältlich:

<http://www.praxisbuch.net>

Im angesprochenen Buch werden auch Funktionen der OpenCV-Bibliothek verwendet, die optional an die IVT-Bibliothek angeschlossen werden kann (vgl. Anhang C und [Bradski 08]). Die OpenCV ist genauso quelloffen wie die IVT, allerdings wesentlich mächtiger. Für den Einsatz auf einem eingebetteten System ist sie aus Gründen des Umfangs nur eingeschränkt geeignet.<sup>7</sup> Es spricht aber technologisch und lizenztechnisch nichts dagegen, interessante Funktionen aus der OpenCV herauszulösen, selbst zu verwenden und auch kommerziell zu nutzen. Die OpenCV steht nicht unter GNU, sondern unter einer dedizierten Lizenz. Hierzu ein Zitat aus [Bradski 08]:

*„Although Intel started OpenCV, the library is and always was intended to promote commercial and research use. It is therefore open and free, and the code itself may be used or embedded (in whole or in part) in other applications, whether commercial or research. It does not force your application code to be open or free. It does not require that you return improvements back to the library—but we hope that you will.“*

---

<sup>7</sup> Tatsächlich hat aber aktuell gerade die Firma Vision Components die OpenCV auf ihre Smart-Camera-Serie portiert. Vgl. <http://www.vision-comp.com>.

Zu Grundlagen zur OpenCV und zum Leistungsumfang vgl. auch die folgenden Quellen:

<http://sourceforge.net/projects/opencvlibrary>

<http://opencvlibrary.sourceforge.net>

<http://tech.groups.yahoo.com/group/OpenCV>

Weiterhin bietet auch das angesprochene, kürzlich erschienene Buch von Gary Bradski und Adrian Kaehler einen guten Einstieg [Bradski 08]. Leseprobe und Quelltextarchiv finden sich unter:

<http://oreilly.com/catalog/9780596516130>



## Ausblick

### 16.1 Communities, Projekte, Trends

In den vorangegangenen Kapiteln wurden zahlreiche Methoden und Implementierungen vorgestellt, die zusammengekommen einen leistungsfähigen Werkzeugkasten ergeben. Damit steht nun der Umsetzung größerer Projekte aus den Bereichen Automatisierungstechnik, Hausautomation, Automotive, Robotik und Wearable nichts mehr im Wege. Zur Ideenfindung und als Hilfestellung haben wir im vorliegenden Abschnitt einen Abriss zusammengefasst, welcher ohne Anspruch auf Vollständigkeit einige interessante Projekte und die Web-Communities vorstellt.

Einige der Communities haben sich auf bestimmte Hardware-Plattformen fokussiert, andere haben sich besonderen Software-Problemen angenommen (z. B. Echtzeitverhalten). Den Gruppen ist gemein, dass sie schnell und unkompliziert kompetente Hilfestellung bieten.

- Embedded-Debian-Project, Community zu einer schlanken Embedded-Variante von Debian GNU/Linux mit dem Ziel, Unterstützung für Prozessoren mit geringen Ressourcen zu bieten:  
<http://www.emdebian.org>
- Embedded-Linux-Projekt zur Portierung des Linux-Kernels für den Betrieb auf Mikrocontrollern:  
<http://www.uclinux.org>
- Internet-Plattform zum Thema uClinux und Embedded Linux (Abbildung 16.1):  
<http://www.ucdot.org>
- Internetauftritt der NSLU2-Gemeinde mit Informationen rund um dieses Gerät:  
<http://www.nslu2-linux.org>

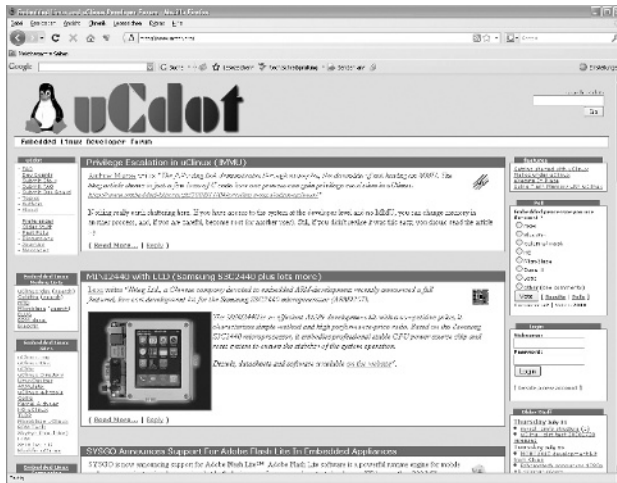


Abb. 16.1. Webauftritt der uCdot-Community.

- Diskussionsplattform zum Thema Real-time-Linux mit einer Auflistung vieler Beispielprojekte:  
<http://www.realtimelinuxfoundation.org>
- Anleitungen, Hilfestellungen und Dokumentationen rund um Embedded Linux:  
<http://www.embeddedlinuxinterfacing.com>
- Website mit ausführlichen Beschreibungen zu Linux-basierten Geräten aller Art:  
<http://linuxdevices.com>

Embedded-Boards sind stromsparender und robuster als herkömmliche PCs. Dadurch eignen sie sich gut für den mobilen Einsatz. Neben einer Verwendung in kommerziellen Produkten wie Mobiltelefonen, PDAs und Navigationsgeräten wird auch im privaten Bereich davon Gebrauch gemacht:

- Wifi-Robot, über WLAN ferngesteuerte WRT54-Roboterplattform:  
<http://www.jbprojects.net/projects/wifirobot/>
- Sammelpunkt für Informationen rund um Linux-basierte Roboter:  
<http://www.linuxrobots.org>

Modularität, hohe Stabilität, Echtzeitfähigkeit und Lizenzfreiheit machen Linux auch für die Automatisierungstechnik interessant. Entsprechend werden mittlerweile viele Automatisierungsaufgaben mit Embedded-Linux-Derivaten umgesetzt. Im relativ jungen Bereich der Hausautomation ist der Einsatz von Linux auch für den privaten Anwender lohnenswert.

- *Open Source Automation Development Lab*, Portal zur Unterstützung der Open-Source-Entwicklung im Bereich der Automatisierungstechnik:  
<http://www.osadl.org>
- Entwickler-Portal für Linux-Systeme in der Prozessautomatisierung:  
<http://www.linux-automation.de>
- OpenRemote-Community für die Hausautomation und Domotik:  
<http://openremote.org>
- xAP-Protokoll zur Vernetzung verschiedener Geräte und Anwendungen in der Hausautomatisierung:  
<http://www.xapautomation.org>
- Steuerung von Heizung und Solaranlage auf Basis eines Linux-Systems:  
<http://www.bastelitis.de/steuerung-von-heizung-und-solaranlage-mit-linux/>

Wurde der Begriff *Open Source* in der Industrie anfangs kritisch betrachtet, so machen sich mittlerweile viele Firmen die Vorteile zunutze und stellen den Entwicklern freie Software-Frameworks oder Toolchains zur Verfügung. Die zugehörigen Embedded-Plattformen sind im Regelfall vergleichsweise günstig zu erwerben:

- OpenMoko: Linux-basiertes Betriebssystem für das Neo FreeRunner-Mobiltelefon. Das Projekt steckt noch in den Kinderschuhen, die Hardware kann aber bereits erworben werden (Abbildung 16.2):  
<http://www.openmoko.com>

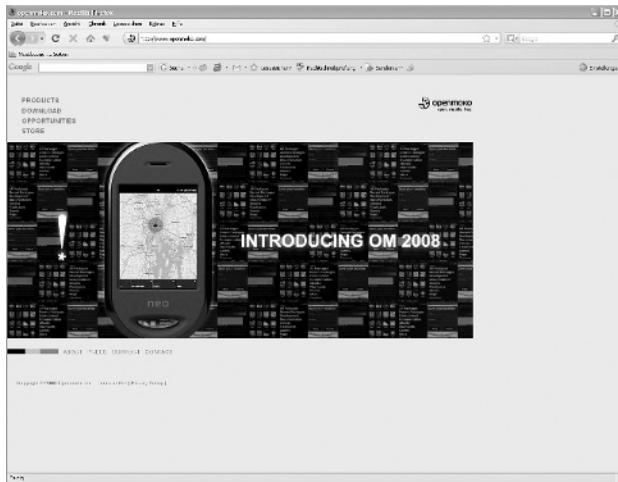


Abb. 16.2. Webauftritt der OpenMoko-Community.

- LeanXCam, eine intelligente Kamera auf der Basis von uClinux und eines freien Software-Frameworks (Abbildung 16.3):  
<http://www.scs-vision.ch/leanxcam/>



Abb. 16.3. Webauftritt des Herstellers der leanXcam.

Sonstige interessante Projekte und Anwendungen:

- LAN/WLAN-Schnittstelle für Digitalkameras auf der Basis der NSLU2:  
<http://webuser.fh-furtwangen.de/~dersch/gphoto/remote.html>
- NSLU2 als Server für wview zur Anzeige von Wetterdaten:  
<http://www.wviewweather.com>
- Mediaserver mit Schnittstelle zu *Roku Soundbridge* und *iTunes*:  
<http://www.fireflymediaserver.org>
- Qt für Embedded Linux mit zugehöriger Greenphone-Plattform:  
<http://trolltech.com/products/device-creation/embedded-linux>  
<http://qtopia.net/modules/devices/>

Abschließend sei ein interessantes Special des Linux-Magazins genannt. Hier werden aktuelle Entwicklungen und Trends vorgestellt und diskutiert:  
<http://www.linux-magazin.de/themengebiete/special/embedded>

## 16.2 Schlusswort und Kontaktdaten

Parallel zum Erscheinen des Buches haben wir eine begleitende Website eingerichtet. Der Zugang kann über die Website des Springerverlages, über die Praxisbuch-Website ([praxisbuch.net](http://praxisbuch.net)) oder auch direkt erfolgen (letzten Schrägstrich nicht vergessen):

<http://www.praxisbuch.net/embedded-linux/>

Auf dieser Website im Download-Bereich werden die Inhalte zum Buch bereitgestellt. Eine besonders aktuelle Version der Quelltext-Dateien kann alternativ auch über Subversion bezogen werden. Die URL zum SVN-Server ist auf der angegebenen Website zu finden. In Anhang F ist die Struktur der Quelltext-verzeichnisse abgebildet.

Wir wünschen allen Lesern viel Spaß und viel Erfolg bei der Umsetzung eigener Ideen und hoffen, dass wir Sie für den hochinteressanten und rasch wachsenden Bereich *Embedded Linux* begeistern konnten.

Bitte melden Sie aufkommende Fragen und Fehler bzw. Unklarheiten. Weiterhin freuen wir uns auch über konstruktive Kritik und über Feedback aller Art.

Joachim Schröder  
Tilo Gockel

<http://www.iain.ira.uka.de>  
[embedded-linux@praxisbuch.net](mailto:embedded-linux@praxisbuch.net)

# A

---

## Kurzreferenzen

### A.1 Einführung

In diesem Anhang werden mehrere stichwortartige Befehlssammlungen rund um die Linuxkonsole vorgestellt und Konfigurationsmöglichkeiten für diverse Dienste erläutert. Die Sammlungen haben Spickzettelcharakter und umfassen Basisbefehlsschatz, Editoren, Netzwerkeinstellungen, Nutzung von Umgebungsvariablen, Umgang mit der Paketverwaltung Aptitude und anderes.

### A.2 Die Linux-Konsole

#### A.2.1 Basisbefehlsschatz

In der Tabelle sind die wichtigsten, immer wiederkehrenden Befehle für den Umgang mit der Linuxkonsole bzw. dem Terminal zusammengestellt. Weiterführende Informationen, auch hinsichtlich Skripterstellung, finden sich unter folgenden Web-Adressen:

<http://www.tim-bormann.de/sammlung-konsolen-befehle-linux>

<http://nafoku.de/t/unix.htm>

<http://rowa.giso.de/german/shell-scripte.html>

<http://help.unc.edu/213>

Befehls- bzw. Programmende für Konsolenprogramme	q <i>oder</i> ESC <i>oder</i> Strg + z <i>oder</i> Strg + y
Compilieren Anm.: Make ohne Parameter verwendet das Make-File aus dem aktuellen Verzeichnis	make make makefile-dateiname make install
Datei löschen	rm rm beispiel.txt
Datei neu anlegen	touch touch beispiel.txt
Datei, Suche nach	find find -name dateiname <i>alternativ</i> locate (u.U. vorher updatedb für die Datenbank) locate dateiname
Dateien kopieren	cp <Quelle> <Ziel> cp -v *.txt /usr/texte
Dateien vergleichen	diff diff -q datei1.dat datei2.dat
Dateien verschieben (move)	mv <Quelle> <Ziel> mv beispiel.txt /usr/beispiel.txt
Dateien, Suche in	grep grep "test" -i beispiel.txt
Dateiinfo (Dateityp)	file file filmdatei.avi
Dateiinhalt seitenweise anzeigen (Ende mit q)	less less xy.txt
Dateisystem-Reparatur (Anm.: auch GParted kennt die Option „Check“)	cfdisk, e2fsck cfdisk /dev/sdb e2fsck -cf /dev/sdb1 e2fsck -p /dev/sdb1
Dateitransfer via http://	wget wget http://www.example.com
Dateizugriffsrechte ändern [ugoa][+-=][rwx] Lese-(r), Schreib- (w), Ausführrecht (x) für Besitzer (u), Gruppe (g), alle anderen (o) oder alle (a) hinzufügen (+), revidieren (-) oder genau setzen (=)	chmod chmod go+w test.txt chmod go+rx tagebuch.txt
Datums- und Zeitausgabe	date
Disk Usage	du -h   more
Diskfree (freier Platz)	df -h
Editoren	vi (Ende mittels “:q!”), geany, nano, gedit, joe
Entpacker (vgl. auch: zip, unzip, gunzip)	tar tar -xzf beispiel.tar.gz tar -xjf beispiel.bz2
Entpacker	/bin/busybox unzip /bin/busybox unzip beispiel-archiv.zip

Tabelle A.1. Basisbefehlsschatz, I.

Environment-Variablen ausgeben	env
Freies RAM / Swap	free
Herunterfahren	halt <i>bzw.</i> shutdown shutdown -h now
Hilfe, Manual Pages bzw. man pages	man befehl
Kalender	cal
Kernelmodule, geladene als Liste ausgeben	lsmod lsmod   grep -i i2c
Kernel-Version	uname -a
Konsole öffnen bzw. schließen	Strg + Alt + F1 ... Strg + Alt + F6 (maximal sechs Konsolen) Strg + Alt + F7: zurück zur grafischen Oberfläche
Konsole, letzte Befehle und Ausgaben löschen	clear
Laufwerke: Infos über eingebundene; neue Laufwerke einbinden bzw. mounten	mount mount /dev/sda1 /mnt mount -t msdos /dev/sda1 /mnt
Laufwerkseinbindung aufheben, bzw. unmounten	umount /dev/sda1
Links bzw. symbolische Verweise erstellen	ln ln -s /docs/projects/beispiel.txt ln -s /docs/projects/beispiel.txt anderername.txt
Mausgeschwindigkeit u.a. xorg-Einstellungen	xset xset m 12/10 0
Message Buffer des Kernels ausgeben (display messages)	dmesg dmesg   less dmesg   grep -i usb
Module zum Kernel hinzufügen (Wrapper für insmod)	modprobe modprobe -v ntfs
Netzwerk-Check: Existiert der Teilnehmer? (Ende mit Strg + y)	ping ping 192.168.1.34
Netzwerk-Info und -einstellungen	ifconfig ifconfig eth0 192.168.0.2 netmask 255.255.255.0
Netzwerk-Kommunikation, „Fernsteuerung“ eines PCs	ssh (secure shell) ssh meinbenutzername@192.168.2.1
Netzwerk-Kommunikation, sichere Dateikopie	scp (secure copy) scp mymusic/*.mp3 schnulli@192.168...:~/musik/
Neustart	reboot
Packages, Ausgabe der installierten (ehemals: Redhat Package Manager)	rpm rpm -qa   grep suchbegriff*
Paketmanager unter Debian	apt-get apt-get install build-essential apt-get install <paket1> <paket2> ... apt-get install libjpeg-dev libsdl-dev apt-get update apt-get upgrade

Tabelle A.2. Basisbefehlsschatz, II.



Partitionieren (HDDs)	GParted <i>oder</i> fdisk fdisk -l
Passwort ändern	passwd
PCI-Geräteliste	lspci lspci -v
Programmpfad ausgeben	which which mozilla
Prozess beenden	kill <Prozess-ID> (die ID ist erhältlich via ps) kill -9 PID (-9: nicht abfangbar) killall -9 prozessname (über Name, statt über PID)
Prozesstabelle	ps (liefert auch die Prozess-ID) ps aux
Screenshot	import import -window root screenshot.png
Sound-Einstellungen	alsamixer
Super-User-Modus (root)	su -
Super-User-Modus zur Programmausführung	sudo sudo fdisk -l sudo apt-get install rdate
Table of Processes: Informationen zu User, Auslastung...; Ende mit q	top
Taschenrechner	bc
Toolsammlung (ping, vi, ...)	bin/busybox
Uhrzeit	date
Umgebungsvariablen	env
USB-Geräteliste	lsusb lsusb -v   grep -i removable
User, Anzeige des aktuellen	whoami
Verzeichnis anlegen	mkdir mkdir test
Verzeichnis löschen	rmdir rmdir test
Verzeichnis, Anzeige des aktuellen	pwd (für: print working directory)
Verzeichnisinhalt ausgeben	dir <i>oder</i> ls ls -la ls /etc > etc-inhalt.txt
Verzeichniswechsel	cd cd .. (eine Ebene höher) cd /usr/bin c - (zuletzt besuchtes Verzeichnis)
Verzeichniswechsel nach Home	cd

**Tabelle A.3.** Basisbefehlsschatz, III.

## A.2.2 Editoren

### Der vi-Editor

Der berühmte und berüchtigte vi-Texteditor<sup>1</sup> gehört schon seit den Anfängen zum Bordwerkzeug von UNIX- und entsprechend auch Linux-Betriebssystemen. Die Bedienung des Editors ist für den verwöhnten Anwender, der nur die Wordstar-Befehle von MS Word u. a. kennt, mehr als ungewöhnlich. Es lohnt sich aber, zumindest einige Basisbefehle zu lernen, um die Vorteile des Editors ab und an nutzen zu können: Er ist ressourcenschonend, schlank, schnell und fast unter jedem Linux verfügbar.

Auch wer nicht mit vi arbeiten möchte, sollte doch zumindest den Befehl zum Verlassen des Programms kennen, falls er einmal vi versehentlich oder aus Neugier startet. Zuerst <Esc> betätigen, um aus einem etwaigen Befehlsmodus herauszukommen, dann "q!", dann mit <Return> abschließen. Zusammengekommen ergibt dies für das Verlassen ohne Speichern:

```
<Esc>
:q! <Return>
```

Der Aufruf von vi geschieht mittels vi <dateiname>. Sollte hierbei die mitgegebene Datei noch nicht existieren, so wird sie neu angelegt. In der Bedienung kennt der vi nun drei verschiedene Eingabemodi:

**Befehlsmodus:** Nach dem Aufruf von vi befindet sich der Editor im Befehlsmodus, d. h. die eingegebenen Zeichen werden nicht als Texteingaben, sondern als Befehle interpretiert. Für eine Texteingabe muss ein Wechsel in den sog. Eingabemodus erfolgen (Bsp.: Befehl i, vgl. auch Tabelle A.4).

**Eingabemodus:** Im Eingabemodus kann eine direkte Texteingabe erfolgen. Mit einer Betätigung der Taste <Esc> gelangt der Anwender zurück in den Befehlsmodus.

**Kommandozeile:** An der Kommandozeile können Dateien geöffnet und gespeichert werden. Auch zum Verlassen des Editors muss man sich in diesem Modus befinden. Zur Kommandozeile gelangt man vom Befehlsmodus durch die Eingabe von ":". Vom Eingabemodus aus ist entsprechend die Eingabe "Esc :" erforderlich.

Die Bewegung im Text erfolgt mit den Cursor-Tasten oder mit den Tasten "l, h, j, k". Wie schon angedeutet, wird der vi im Befehlsmodus mit dem Befehl "q!", im Eingabemodus entsprechend mit "Esc:q!" beendet.

Zu den weiterführenden Befehlen gibt Tabelle A.4 eine Übersicht.<sup>2</sup>

<sup>1</sup> vi steht für Visual Interface, eine Bezeichnung, die den Unterschied zu rein zeilenorientierten Editoren wie ex oder edlin herausstellen soll.

<sup>2</sup> Die Befehle sind teilweise der Kurzanleitung entnommen, vgl. [http://www.my-space.li/schule/editor\\_VI.pdf](http://www.my-space.li/schule/editor_VI.pdf).

<b>Eingabemodus</b>	
Einfügen, rechts vom Cursor (append)	a
Anhängen am Zeilenende (Append)	A
Einfügen, links vom Cursor (insert)	i
Einfügen, am Zeilenanfang (Insert)	I
In neuer Zeile danach einfügen (open)	o
In neuer Zeile davor einfügen (Open)	O
Zeichen am Cursor ersetzen (substitute)	s
Ganze Cursorzeile ersetzen (Substitute)	S
Einschalten des Überscheibe-Modus (Replace)	R
Überschreiben des nächsten Wortes (change word)	cw
Ersetzen der ganzen Zeile (change)	cc
Ersetzen des Rests der Zeile (Change)	C

<b>Löschen und Kopieren</b>	
Löschen des nächsten Wortes (delete word)	dw
Löschen der nächsten n Wörter (n delete words)	ndw
Löschen der aktuellen Zeile	dd
Löschen der nächsten n Zeilen	ndd
Löschen ab Cursor bis Dateiende	dG
Löschen der restlichen Zeile	D
Löschen des Zeichens an der Cursorposition	x
Löschen der nächsten n Zeichen von der Cursorpos.	nx
Löschen des Zeichens vor der Cursorposition	X
Kopieren der aktuellen Zeile in den Puffer	yy
Kopieren der nächsten n Zeilen in den Puffer	nyy
Kopieren des Zeilenrestes ab Cursorposition in ...	y\$
Kopiere Inhalt des Puffers unter die akt. Zeile (paste)	p
Kopiere Inhalt des Puffers über die akt. Zeile (Paste)	P
Kopiere Datei „file“ unter die aktuelle Zeile	:r file
Einrücken des Textes bis zur zugehörigen Klammer	>%

<b>Änderungen rückgängig machen</b>	
Letzte Änderung rückgängig (undo)	u
Änderung in der aktuellen Zeile rückgängig (Undo)	U
Verwerfen aller Änderungen seit dem letzten Sichern	:e!
Verlassen ohne Sichern (quit)	:q!

<b>Sichern und Beenden</b>	
Sichern und verlassen	ZZ
dito (write and quit)	:wq!
Verlassen ohne sichern (quit)	:q!
Sichern (write)	:w
Sichern der Daten in Datei „file“ (write file)	:w file

Tabelle A.4. Die wichtigsten vi-Befehle.

## Der nano-Editor

Neben dem Veteranen `vi` ist auch der etwas komfortablere Editor `nano` mittlerweile unter vielen Linux-Distributionen verfügbar. Wie `vi`, so ist auch `nano` ein sog. Kommandozeileneditor und kann entsprechend nicht nur am Prompt der Kommandozeile, sondern auch am Prompt einer Telnet- oder SSH-Session ausgeführt werden. Auch dieser Editor besitzt keine grafische Benutzerschnittstelle wie die großen Brüder `geany` oder `gedit`. Dennoch fällt dem Einsteiger der Umgang damit wesentlich leichter als mit `vi`, da der Editor zumindest in Ansätzen die sog. Wordstar-Kommandos beherrscht.<sup>3</sup>

Im Gegensatz zu `vi` ist `nano` zwar unter vielen, aber nicht unter allen Linux-Systemen automatisch verfügbar. Der Editor kann aber relativ einfach nachinstalliert werden. Unter Debian-basierten Systemen kann hierzu die Paketverwaltung genutzt werden:

```
sudo apt-get install nano
```

Unter anderen Linux-Systemen müssen die Quelltexte geladen und kompiliert werden:

```
wget "http://www.nano-editor.org/dist/v2.0/nano-2.0.6.tar.gz"
tar -xzf nano-2.0.6.tar.gz
cd nano-2.0.6
./configure
make
make install
cd ..
rm -r nano*
```

Wie bereits bei `vi`, so kann auch bei `nano` eine Textdatei beim Aufruf des Editors mitgegeben werden:

```
nano dateiname
```

Sollte die Textdatei noch nicht existieren, so wird sie neu angelegt. In der Statuszeile des Programms findet sich eine Hilfestellung zu den wichtigsten Befehlen (Abbildung A.1). Hierin steht das Caret-Zeichen für die Taste `<Strg>` und so ist z. B. mit `<Strg> + <G>` auch eine Hilfe verfügbar.

Die Navigation im Text erfolgt wie gewohnt mit den Pfeiltasten, und auch sonst ist die Bedienung von `nano` relativ einfach. Vorsicht ist allerdings an einer Stelle geboten: `nano` fügt beim Laden von Textdateien u. U. automatisch zusätzliche Zeilenumbrüche ein. Zumindest bei Konfigurationsdateien ist dies nicht gewünscht, diese sind zu laden wie folgt:

```
nano -w dateiname
```

<sup>3</sup> Vgl. auch Doku und Download unter <http://www.nano-editor.org>.



Abb. A.1. Screenshot zum nano-Editor unter Puppy Linux.

## A.3 Netzwerkeinstellungen und SSH

### A.3.1 Netzwerkeinstellungen

Die aktuelle Konfiguration der Netzwerkschnittstellen kann unter Linux mit dem Befehl `ifconfig` ausgegeben werden:

```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:18:39:36:xx:xx
          inet addr:192.168.1.160  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::218:39ff:fe36:xxxx/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:83487 errors:0 dropped:0 overruns:0 frame:0
          TX packets:92765 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:16875535 (16.0 MiB)  TX bytes:42182602 (40.2 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:97 errors:0 dropped:0 overruns:0 frame:0
          TX packets:97 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:6565 (6.4 KiB)  TX bytes:6565 (6.4 KiB)
```

Bei einer NSLU2 zeigt die Ausgabe zwei Netzwerkschnittstellen `eth0` und `lo`. Bei `eth0` handelt es sich um eine physikalische Netzwerkschnittstelle, die im vorliegenden Fall über einen integrierten Netzwerk-Baustein realisiert ist. Der in der ersten Zeile aufgeführte Eintrag `HWaddr` stellt die physikalische MAC-Adresse des Schnittstellenbausteins dar, danach folgt die aktuelle Konfiguration. `lo` stellt eine sog. Loopback-Schnittstelle oder Schleifenschaltung dar. Diese ist auf jedem Unix-System vorhanden und hat die besondere Eigenschaft, dass Sender und Empfänger identisch sind. Für Loopback-Netzwerke ist der Adressraum `127.xxx.xxx.xxx` reserviert und üblicherweise

wird 127.0.0.1 als Loopback-Schnittstelle verwendet. Dieser Mechanismus wird von Client- und Serverprogrammen benötigt, die auf dem gleichen Rechner laufen, aber nur über das Netzwerk kommunizieren können. Die Daten werden hierbei wie herkömmliche Netzwerkdaten unter Beteiligung mehrerer Schichten übertragen. Entsprechend ist dies eine oft verwendete Möglichkeit, die Funktion des TCP/IP-Stacks zu überprüfen. Die Netzwerk-Schnittstellen lassen sich über folgenden Befehl abschalten:<sup>4</sup>

```
$ ifdown eth0
```

Der Befehl `ifup` aktiviert die Schnittstelle mit den in der Konfigurationsdatei `/etc/network/interfaces` hinterlegten Einstellungen.<sup>5</sup> Bei der Ausführung in einer SSH-Shell ist zu beachten, dass ein Abschalten der Netzwerkverbindung den Benutzer ausschließen würde. Hier hilft der folgende Aufruf, bei dem das Netzwerk unverzüglich wieder aktiviert wird:

```
$ ifdown eth0 && ifup eth0
```

Um der Netzwerkschnittstelle `eth0` eine statische IP 192.168.1.20 zuzuweisen und für den Betrieb in einem Subnetz 192.168.1.(0-255) mit Router 192.168.1.1 zu konfigurieren, sind folgende Netzwerkeinstellungen zu hinterlegen:

```
iface eth0 inet static
    address 192.168.1.20
    netmask 255.255.255.0
    gateway 192.168.1.1
    up echo Interface $IFACE going up
    down echo Interface $IFACE going down
```

Über die beiden letzten Zeilen können Befehle aufgelistet werden, die beim An- (`up`) oder Abschalten (`down`) der Schnittstelle ausgeführt werden. Informationen über den *Domain-Name-Server* (DNS) werden in der Datei `/etc/resolv.conf` hinterlegt:

```
$ cat /etc/resolv.conf
nameserver 192.168.1.1
```

Üblicherweise übernimmt diese Aufgabe in einem Hausnetzwerk ebenfalls der als *Gateway* eingetragene Router. Ist das Paket `resolveconf` unter Debian installiert, so lassen sich die Nameserver-Einstellungen durch folgende Zeile zentral in `/etc/network/interfaces` hinterlegen:

```
dns-nameservers 192.168.1.1
```

<sup>4</sup> Dies funktioniert, sofern das Paket `ifupdown` verwendet wird. Alternativ kann dies auch über `ifconfig eth0 inet down` (`up`) geschehen.

<sup>5</sup> Unter Ubuntu können Netzwerkeinstellungen komfortabel über den Menüpunkt *Systemverwaltung / Netzwerk* vorgenommen werden. Ein OpenWrt-System bezieht die Netzwerkeinstellungen aus `/etc/config/network`.

Soll die Konfiguration der Netzwerkschnittstelle nicht mit einer statischen IP, sondern mittels DHCP erfolgen, so reicht folgender Eintrag in `/etc/network/interfaces` aus:

```
iface eth0 inet dhcp
```

Dies setzt voraus, dass ein DHCP-Client vorhanden ist. Unter Debian wird üblicherweise `dhcpc3-client` verwendet, OpenWrt benutzt `udhcpc`. In der Datei `/etc/hostname` ist der Name hinterlegt, unter dem der Rechner im Netzwerk auftaucht. In `/etc/hosts` können Aliase für Rechner im Netzwerk angelegt werden, um diese mit einer hinterlegten Abkürzung ansprechen zu können. Der folgende Eintrag bewirkt, dass der Rechner `google.de` mit IP `216.239.59.104` über die Kurzform `go` erreicht werden kann:

IP-Adresse	Rechnername	Kurzform
216.239.59.104	google.de	go

Für WLAN-Schnittstellen werden spezielle Einstellungen mit dem Programm `iwconfig` vorgenommen, welches im Paket `wireless-tools` enthalten ist. Weitere Informationen zur Linux-Netzwerkkonfiguration liefern <http://www.linux-related.de/index.html?/admin/netzwerk.htm> und <http://www.debian.org/doc/manuals/reference/ch-gateway.de.html>.

### A.3.2 Secure Shell

Das SSH-Protokoll (Secure Shell) ermöglicht verschlüsselte Netzwerkverbindungen zwischen Rechnern und wird für die Absicherung einer entfernten Rechnerverwaltung, eine sichere Datenübertragung und für das *Tunneln* unsicherer Protokolle verwendet. Die auf SSH basierenden Werkzeuge `ssh`, `scp` und `sftp` ersetzen dabei die herkömmlichen, unsicheren Unix-Tools `telnet`, `rcp` und `ftp`.

Mit OpenSSH<sup>6</sup> steht eine freie Sammlung der SSH-Werkzeuge zur Verfügung, die unter Debian im Paket `openssh-client` enthalten ist. Ein SSH-Server kann über das Paket `openssh-server` installiert werden. Unter OpenWrt ist OpenSSH ebenfalls verfügbar, hier bietet sich jedoch die Verwendung der schlankeren Version *Dropbear*<sup>7</sup> an. Die von OpenSSH etwas abweichende Bedienung ist in Abschnitt 3.3 ausführlich erklärt.

Die Einstellungen für SSH-Client und -Server werden in den Dateien `/etc/ssh/ssh_config` und `/etc/ssh/sshd_config` hinterlegt. Hierzu zählen Angaben wie SSH-Portnummer (üblicherweise Port 22), Authentifizierungsbeschränkungen, X11-Handhabung und Logging. Erläuterungen zu den einzelnen

<sup>6</sup> Vgl. <http://www.openssh.org>.

<sup>7</sup> Enthalten im `ipkg`-Paket `dropbear`.

Variablen sind in den Manpages zu `ssh_config` und `sshd_config` zu finden. Das Einloggen via SSH auf einem anderen Rechner erfolgt unter Angabe des Benutzers und des Rechnernamens:

```
$ ssh <name>@<host>
```

Bei einem ersten Verbindungsaufbau zu einem Host fragt der Client üblicherweise nach, ob tatsächlich eine SSH-Verbindung zu diesem unbekannten Rechner aufgebaut werden soll. Nach einer Bestätigung mit `yes` wird der RSA-Fingerabdruck in die Liste der bekannten Host-Rechner übernommen.<sup>8</sup> Als Passwort ist das für den Benutzer auf dem Hostsystem verwendete Kennwort anzugeben. Mit dem Programm `scp` lassen sich Dateien nach folgendem Schema verschlüsselt auf einen Host-Rechner übertragen:

```
$ scp test.txt <name>@<host>:<directory>
```

Die Daten werden auf dem Host-System im Verzeichnis `<directory>` abgelegt. Erfolgt keine Angabe, dann wird standardmäßig das Home-Verzeichnis von `<name>` verwendet. Über die Option `scp -r <name>...` lassen sich komplette Verzeichnisse rekursiv kopieren. Die wiederholte Eingabe der Passwörter, wie sie z. B. bei Verwendung eines Cross Compilers zum Test des Compilats notwendig ist, wird schnell lästig. Um dem Anwender die Eingabe zu ersparen, können mit der Hinterlegung von SSH-Schlüsseln dauerhaft sichere Rechnerpaarungen definiert werden. Diese Zugriffskontrolle basiert auf der sog. Public-Key-Authentifizierung. Bei dieser Methode wird zunächst auf dem Client ein SSH-Schlüsselpaar erzeugt:

```
$ ssh-keygen -t rsa
```

Schlüssel vom Typ `rsa` sind etwas weniger aufwändig in der Berechnung als `dsa`, aber dennoch sehr sicher. Wahlweise kann ein Passwort zur Verschlüsselung des privaten Schlüssels festgelegt werden. Als Ergebnis werden ein privater Schlüssel `id_rsa` und ein öffentlicher Schlüssel `id_rsa.pub` erzeugt, die beide in `~/.ssh/` auf dem erzeugenden Rechner abgelegt werden sollten. Der private Schlüssel wird verwendet, um Daten zu entschlüsseln, welche zuvor mit dem öffentlichen Schlüssel verschlüsselt wurden. Dieser Schlüssel muss unbedingt geheim gehalten und sollte nie über das Netzwerk kommuniziert werden. Der öffentliche Schlüssel kann für eine Authentifizierung verwendet werden, indem er auf einen Hostrechner kopiert und dort der Datei `authorized_keys` hinzugefügt wird:

```
$ scp id_rsa.pub <name>@<host>:
$ ssh <name>@<host>
name@host's password: *****
$ cat id_rsa.pub >> ~/.ssh/authorized_keys
```

<sup>8</sup> Genauer gesagt in die Datei `~/.ssh/known_hosts` jenes Anwenders, in dessen Shell der Vorgang stattfindet. Neben der Datei `known_hosts` werden in diesem Verzeichnis weitere benutzerspezifische Einstellungen abgelegt.



In dieser Datei werden alle öffentlichen Schlüssel von Benutzern anderer Rechnern gesammelt, die sich ohne Passworтеingabe über SSH einloggen dürfen. Unverschlüsselte Daten lassen sich über eine SSH-Verbindung *tunneln* und können damit sicher übertragen werden. Mit dem Prinzip des *Local Port Forwarding* wird zwischen einem lokalen Port des Clients und einem entfernten Port des Hosts eine verschlüsselte Verbindung aufgebaut. Im vorliegenden Fall erfolgt dies vom lokalen Port 64 000 auf den Remote-Port 64 001 des Rechners <host>:

```
$ ssh -L 64000:127.0.0.1:64001 <name>@<host>
```

Daten können nun z. B. mittels folgendem Befehl vom Client-System gesendet werden:

```
$ echo dies ist ein test | nc 127.0.0.1 64000
```

Sie tauchen wiederum in der Shell des Hostrechners auf:

```
$ nc -l -p 64001
dies ist ein test
```

Analog lässt sich ein Port des Host-Systems über *Remote Port Forwarding* mit der Option `-R` auf das Client-System umleiten bzw. „holen“. Wird folgender SSH-Tunnel vom Client aus erstellt, so gelangen alle auf Port 64 000 des Host-Rechners geschriebenen Daten automatisch auf Port 64 001 des Client-Systems:

```
$ ssh -R 64000:127.0.0.1:64001 <name>@<host>
```

Sowohl beim Local- als auch beim Remote-Port-Forwarding verbleiben die Daten in beiden Beispielaufrufen auf den empfangenden Rechnern (127.0.0.1). Hier könnte auch ein anderer Rechner aus dem jeweiligen lokalen Netz angegeben werden. Eine tiefergehende Einführung in OpenSSH ist unter <http://suso.org/docs/shell/ssh.sdf> zu finden. Die URL [http://www.qcnetwork.com/vince/doc/divers/udp\\_over\\_ssh\\_tunnel.html](http://www.qcnetwork.com/vince/doc/divers/udp_over_ssh_tunnel.html) zeigt die Weiterleitung von UDP-Daten über einen SSH-Tunnel, wie sie für Kapitel 13 Anwendung finden könnte.

## A.4 Weitere Werkzeuge und Dienste

### A.4.1 Paketverwaltung APT

Das *Advanced Packaging Tool (APT)* wird als Paketverwaltungssystem für Debian eingesetzt und dient dazu, Pakete auf Quellrechnern zu suchen, herunterzuladen, zu installieren und Abhängigkeiten aufzulösen.<sup>9</sup>

<sup>9</sup> Die bei OpenWrt eingesetzte Paketverwaltung `ipkg` wird in Kapitel 3 genauer erklärt.

Weiterhin lässt sich ein System sehr einfach auf einem definierten Stand halten, was insbesondere für die Administration mehrerer Rechner hilfreich ist. Unter Debian werden die Pakete in Form von **.deb**-Dateien installiert. Mögliche Quellen für diese Pakete sind in der Datei `/etc/apt/sources.list` hinterlegt, die für eine NSLU2 unter Debian folgendermaßen aussehen kann:

```
deb http://ftp.de.debian.org/debian/ etch main
deb-src http://ftp.de.debian.org/debian/ etch main

deb http://security.debian.org/ etch/updates main
deb-src http://security.debian.org/ etch/updates main
```

Die Schlüsselwörter **deb** bzw. **deb-src** beschreiben den Umstand, dass es sich um einen Server mit vorkompilierten Binärpaketen bzw. um einen Server mit Programmquellen handelt. Nach dem Link folgen die Namen der Distribution und deren installierte Komponenten. Wurde das System von einer CD aus installiert, so findet sich oft ein Eintrag der Form **deb cdrom:xy**. Falls die CD nach der Installation nicht mehr verfügbar ist, so bereitet dieser Eintrag Probleme und kann einfach auskommentiert werden. Eine Liste aller Debian-Server findet sich unter [http://www.debian.org/mirror/mirrors\\_full](http://www.debian.org/mirror/mirrors_full). Aus der APT-Programmbibliothek kommt hauptsächlich das Programm **apt-get** zum Einsatz, das mit folgenden Parametern aufgerufen wird:

**install <paketname>**: Installiert ein Paket und dessen Abhängigkeiten.

**remove <paketname>**: Entfernt ein Paket und davon abhängende Pakete, belässt aber die Konfigurationsdateien.

**purge <paketname>**: Entfernt ein Paket, davon abhängende Pakete und die Konfigurationsdateien.

**upgrade**: Aktualisiert alle Pakete in der aktuellen Distribution.

**dist-upgrade**: Aktualisiert das gesamte System auf die neueste Distribution.

**update**: Holt die neuesten Datenbankinformationen vom Debian-Server.

Für eine Aktualisierung der Pakete innerhalb einer Distribution kommt folgende Kombination zum Einsatz:

```
$ apt-get update
$ apt-get upgrade
```

Wenn von einem Paket nur ein Teil des Namens oder der Beschreibung bekannt ist, so kann mit dem Befehl **apt-cache** in der Datenbank nach dem vollständigen Namen gesucht werden. Als Beispiel gibt folgender Aufruf alle Datenbankeinträge zurück, die das Suchwort **openssh** enthalten:

```
$ apt-cache search openssh
connect-proxy - Establish TCP connection using SOCKS4/5 and HTTP tunnel
keychain - key manager for OpenSSH
libssl0.9.7 - SSL shared libraries
libssl0.9.8 - SSL shared libraries
openssh-blacklist - list of blacklisted OpenSSH RSA and DSA keys
```

```
openssh-client - Secure shell client, an rlogin/rsh/rcp replacement
openssh-server - Secure shell server, an rshd replacement
...
```

In Kombination mit den Optionen **show** und **depends** lassen sich mit **apt-cache** auch Paketinformationen und Abhängigkeiten anzeigen. Der Befehl **dpkg-query --list** listet alle bereits installierten Pakete auf, sodass in Kombination mit **grep** nach Schlüsselwörtern gesucht werden kann:

```
$ dpkg-query --list | grep openssh
openssh-client      4.3p2-9      Secure shell client, an rlogin/rsh/rcp repla
openssh-server      4.3p2-9      Secure shell server, an rshd replacement
```

Unter Ubuntu wird APT in Kombination mit dem grafischen Frontend *Synaptic* verwendet. Ein umfangreiches APT-HowTo findet sich unter <http://www.debian.org/doc/manuals/apt-howto/>.

### A.4.2 Umgebungsvariablen

Umgebungsvariablen werden unter Linux verwendet, um bestimmte Werte, Namen oder Suchpfade in Form von Variablen zu hinterlegen. Das Betriebssystem und die Anwendungen kommunizieren oftmals Einstellungsdaten über Systemvariablen. Eine Variable kann durch folgende Zuweisung erzeugt und mit einem Wert belegt werden:

```
$ TEST=1
$ echo $TEST
1
```

Der Befehl **echo** liefert in Kombination mit dem **\$**-Zeichen den Variableninhalt zurück. Um die Variable neben der Shell auch den Programmen verfügbar zu machen, die in einer Shell aufgerufen werden, ist der Befehl **export** zu verwenden:

```
$ export TEST=1
```

Der Befehl **unset TEST** löscht die Umgebungsvariable wieder. Die wohl wichtigste Verwendung der Systemvariablen ist die Definition des globalen Suchpfades in der Umgebungsvariablen **PATH**. Der folgende Befehl gibt den Inhalt von **PATH** aus:<sup>10</sup>

```
$ printenv PATH
/usr/local/bin:/usr/bin:/bin:/usr/games:
```

Der Befehl **set | less** zeigt alle auf dem System gesetzten Umgebungsvariablen mit Inhalt an. Oftmals ist es notwendig, den Suchpfad zu erweitern,

<sup>10</sup> Allgemein gibt **printenv** den Inhalt von Variablen aus, die zuvor mittels **export** definiert wurden.

um eigene Anwendungen ohne Pfadangabe ausführen zu können. Wichtig hierbei ist, dass die Variable `PATH` nicht einfach überschrieben werden sollte. Als Beispiel wird das Verzeichnis `/usr/local/my_directory` folgendermaßen zum Suchpfad hinzugefügt:

```
PATH=$PATH:/usr/local/my_directory
export PATH
```

Einzelne Pfade werden durch einen Doppelpunkt voneinander getrennt. Entscheidend für eine Verwendung ist, wo die Variablen definiert wurden, da hiervon der Gültigkeitsbereich abhängt. Die in einer Shell manuell gesetzten Variablen sind nur in der Shell dieses Benutzers und den darin erzeugten Kindprozessen gültig. Um Variablen automatisch oder systemweit zu setzen, bietet sich ein entsprechender Eintrag in einer der folgenden Dateien an:

**/etc/profile:** Globale Konfigurationsdatei für systemweite Umgebungsvariablen; sie wird beim Systemstart ausgeführt.

**~/profile:** Lokale, benutzerspezifische Konfigurationsdatei für spezielle Umgebungsvariablen. Die Datei wird beim Einloggen eines Benutzers ausgeführt.

**~/bashrc:** Konfigurationsdatei für die Bash-Shell, wird beim Öffnen eines Bash-Terminals ausgeführt.

### A.4.3 Erstellung von Gerätedateien mit `mknod`

Mit dem Befehl `mknod` werden Gerätedateien für FIFO-Speicher, zeichen- und blockorientierte Geräte erzeugt. Zur Erstellung muss die Major-Nummer des Gerätetreibers bekannt sein, welcher für die Gerätedatei zuständig sein soll. Um mehrere Geräte unterscheiden zu können, ist weiterhin die Angabe einer Minor-Nummer notwendig (vgl. Abschnitt 11.4.1). Die Syntax des Befehls lautet wie folgt:

```
$ mknod -m <mode> <filename> <type> <major> <minor>
```

Über `<mode>` werden die Zugriffsrechte auf die Datei entsprechend einer drei- oder vierstelligen Oktalzahl festgelegt (vgl. Abschnitt A.4.4). Anstelle von `<filename>` wird der Name des Gerätes angegeben. `<type>` enthält die Information darüber, ob es sich um ein zeichenorientiertes Gerät (c), blockorientiertes Gerät (b) oder eine FIFO-PIPE (p) handelt. Eine weitere Information ist, welcher Treiber mit Nummer `major` für dieses Gerät mit Nummer `minor` zuständig sein wird. Ein Aufruf von `cat /proc/devices` zeigt eine Liste der vergebenen Major-Nummern mit zugehörigen Treibernamen an.

Folgender Aufruf erzeugt ein zeichenorientiertes Gerät mit Treibereinsprungspunkt `/dev/my_device`, welches den Treiber mit Major-Nummer 64 verwendet:

```
$ mknod -m 666 /dev/my_device c 64 2
```

Hierbei wird davon ausgegangen, dass bereits zwei Gerätedateien mit Minor-Nummern 0 und 1 vorhanden sind und Lese- und Schreibrechte für alle Benutzer vergeben werden sollen.

#### A.4.4 Zugriffsrechte

Da Linux ein Mehrbenutzersystem ist, muss festgelegt werden, auf welche Verzeichnisse und Dateien bestimmte Benutzer in welcher Art zugreifen dürfen. Diese Festlegung geschieht durch das Setzen von Zugriffsrechten, wobei man hierbei auch vom *Modus* spricht. Der folgende Befehl zeigt die Zugriffsrechte für die Dateien und Verzeichnisse innerhalb des aktuellen Verzeichnisses an:

```
$ ls -l
total 44
drwxr-xr-x 24 joachim joachim 20480 Nov 18 01:25 joachim
drwxr-xr-x  2 tilo      tilo      4096 Jan 13  2008 tilo
```

Die Zugriffsrechte werden durch die ersten zehn Zeichen beschrieben. Nach der Angabe im ersten Zeichen, ob es sich um ein Verzeichnis (d), einen Link (l) oder eine Datei (-) handelt, folgen drei rwx-Blöcke, die die Zugriffsrechte für den Besitzer, die Gruppe und andere Benutzer festlegen. Innerhalb eines rwx-Blocks bedeuten gesetzte Werte an den entsprechenden Stellen, dass Leserechte (r), Schreibrechte (w) oder Ausführungsrechte (x) vorliegen. Bestehen für eine Datei die Rechte `-rwxr--r--`, so darf diese von allen Benutzern gelesen, aber nur vom Besitzer geschrieben und ausgeführt werden. In der obigen Ausgabe folgt nach den Zugriffsrechten in der nächsten Spalte die Anzahl an Hardlinks, welche auf die jeweilige Datei oder das Verzeichnis zeigen, dann der Name des Besitzers und der zugeordneten Gruppe. Weiterhin wird die Größe angegeben, das Datum der letzten Änderung und der Datei- oder Verzeichnisname.

Der Befehl `chmod` (Change Mode) dient dazu, die Zugriffsrechte für eine Datei oder ein Verzeichnis zu ändern. Grundsätzlich kann nur der Besitzer oder der Root-Benutzer Zugriffsrechte festlegen. Folgender Aufruf fügt der Datei `<name>` Lese- und Schreibrechte für alle Benutzer hinzu. Der Modus wird hierbei als Buchstabenkennung übergeben:

```
$ chmod a+rw <name>
```

Sollen die Zugriffsrechte nur in einem bestimmten rwx-Block geändert werden, so wird an Stelle von `a` entweder `u` (Besitzer), `g` (Gruppe) oder `o` (Andere) angegeben. Kombinationen sind ebenfalls möglich. Danach erfolgt die Festlegung, ob Zugriffsrechte hinzugefügt (+), entzogen (-) oder gesetzt (=) werden sollen bevor die eigentlichen Rechte aufgelistet werden. Neben den Werten `r`, `w` und `x` können Dateien auch Sonderrechte wie `s` oder `t` besitzen. Ein `s`-Bit kann nur für Binärdateien gesetzt sein und bewirkt, dass als Benutzerkennung des Prozesses nicht jene des Datei Benutzers, sondern die des Besitzers verwendet wird. Der Prozess wird quasi „im Auftrag“ ausgeführt. Ist für ein Verzeichnis

das **t**-Bit gesetzt, so ist es dessen Besitzer verboten, Dateien anderer Benutzer aus diesem Verzeichnis zu löschen. Dieses Bit wird daher auch als *Sticky*-Bit bezeichnet. Handelt es sich bei **<name>** um ein Verzeichnis, so kann der Befehl über die Option **-R** rekursiv angewandt werden, um die Zugriffsrechte auch für die enthaltenen Dateien und Verzeichnisse anzupassen:

```
$ chmod -R a+rw <name>
```

Neben der Angabe einer Buchstabenkennung kann der Modus auch als drei- oder vierstellige Oktalzahl angegeben werden. Die letzten drei Ziffern der Zahl setzen jeweils die Rechte für Besitzer, Gruppe und Andere, wobei jede Ziffer aus der Summe der Zugriffsrechte für Lesen (4), Schreiben (2) und Ausführen (1) gebildet wird. Bei der Angabe von vier Ziffern werden in der ersten Ziffer spezielle Ausführungsmodi gespeichert, die sich aus dem **t**-Bit (1) und den **s**-Bits für Gruppe (2) und Besitzer (4) zusammensetzen.

Die bisher genannten Beispiele zu den **rwX**-Parametern bieten sich an, wenn einzelne Zugriffsrechte unabhängig von den bestehenden Rechten zu setzen sind. Der Vorteil der Oktal-Kodierung hingegen ist die einfache Repräsentation als Ganzzahl. Diese absolute Darstellung wird von vielen Programmen, Bibliotheken und Skripten verwendet, um Zugriffsrechte anzugeben. Neu angelegte Dateien oder Verzeichnisse enthalten übrigens stets den mit **umask** festgelegten Zugriffsmodus:

```
$ umask
0022
```

Über den Aufruf **umask <wert>** kann die Standardeinstellung überschrieben werden. Bei Aufruf in einer Shell gelten die Änderungen nur dort und nur während dieser Session. Der Zugriff auf Hardware-Schnittstellen wird vom System üblicherweise streng limitiert, aber auch dies lässt sich mit **chmod** anpassen. So ermöglicht als Beispiel der nachfolgende Befehl vollen Zugriff auf die serielle Schnittstelle **/dev/ttyS0** (der Aufruf muss als Root erfolgen):

```
$ chmod 0666 /dev/ttyS0
```

Sollte es notwendig werden, den Besitzerstatus (Besitzer, Gruppe) einer Datei zu ändern, so kann dies mit dem Befehl **chown** erfolgen. Mit folgendem Aufruf wird der Datei **<name>** ein neuer Besitzer **<user>** und eine neue Gruppe **<group>** zugewiesen:

```
$ chown <user>:<group> <name>
```

Eine Änderung des Besitzerstatus' kann nur vom Superuser **root** durchgeführt werden. Mit der Option **-R** werden Besitzer- und Gruppenzugehörigkeit für alle Dateien im Verzeichnis **<name>** geändert.

Mithilfe des Programms **udev** erzeugt und verwaltet der Kernel Gerätedateien. **udev** kommt bereits beim Systemstart zum Einsatz, um Gerätedateien für

serielle Schnittstellen oder für den I<sup>2</sup>C-Bus zu erstellen. Aber auch im Betrieb wird **udev** verwendet, um Hotplugging-Ereignisse auszuwerten. So werden Gerätedateien für mobile Datenträger bspw. erst beim Einstecken dynamisch erzeugt. Name und Zugriffsberechtigungen dieser Geräte sind in Form sog. **udev**-Regeln im Verzeichnis `/etc/udev/rules.d/` hinterlegt.<sup>11</sup>

Um Zugriffsrechte für Gerätedateien automatisch auf einen definierten Modus zu setzen, sind die Einträge entsprechend anzupassen. Die Anpassung erspart dann auch eine erneute Einstellung via **chmod** nach dem Einstecken. Die I<sup>2</sup>C-Gerätedatei `/dev/i2c-0` einer NSLU2 wird über folgende Änderung in `/etc/udev/devfs.rules` für alle Anwender zum Lesen und Schreiben freigegeben:<sup>12</sup>

<code>#KERNEL=="i2c-[0-9]*",</code>	<code>NAME="i2c/%n"</code>
<code>KERNEL=="i2c-[0-9]*",</code>	<code>NAME="i2c/%n", MODE="0666"</code>

Falls unklar ist, in welcher der Einzeldateien die Regel platziert werden soll, so kann über **fgrep** ein bestehender Eintrag gesucht werden:

```
$ sudo fgrep -r "i2c" /etc/udev/
/etc/udev/compat-full.rules:KERNEL=="i2c-[0-9]*", SYMLINK+="%k"
/etc/udev/compat.rules:#KERNEL=="i2c-[0-9]*", SYMLINK+="%k"
/etc/udev/devfs.rules:KERNEL=="i2c-[0-9]*", NAME="i2c/%n"
```

Ein weiterer Vorteil eigener **udev**-Regeln ist die Unabhängigkeit von der Einschaltreihenfolge bei Massenspeichern. USB-Sticks oder Festplatten können anhand ihrer Seriennummer identifiziert und mit einem festgelegten Gerätenamen eingehängt werden. Weiterführende Informationen zu **udev** finden sich unter [http://www.reactivated.net/writing\\_udev\\_rules.html](http://www.reactivated.net/writing_udev_rules.html).

#### A.4.5 Root-Rechte mit **sudo**

Mit dem Programm **sudo** kann ein Befehl mit Root-Rechten ausgeführt werden ohne, dass der Benutzer sich dazu als **root** anmelden muss. Da die umfangreichen Root-Berechtigungen dann nur für die Ausführung des Programms oder Befehls gelten, ist diese Variante sehr viel sicherer als eine meist länger andauernde Anmeldung als Benutzer **root**.<sup>13</sup> Die potenzielle Gefährdung des Systems wird dadurch so gering wie möglich gehalten. Das Programm **sudo** ist unter Debian im gleichnamigen Paket enthalten. Außer **root** dürfen zunächst keine anderen Benutzer von dieser Möglichkeit Gebrauch machen. Um einen Benutzer für die Verwendung von **sudo** freizuschalten, muss dieser in die Datei `/etc/sudoers` eingetragen werden. Hier wird definiert, welche Benutzer

<sup>11</sup> Diese können je nach Installation auch im Verzeichnis `/etc/udev/` liegen.

<sup>12</sup> Standardmäßig erzeugt **udev** Geräte mit Zugriffsmodus 0660.

<sup>13</sup> Sollte es notwendig sein, sich als Benutzer **root** anzumelden, so kann dies von jedem anderen Benutzer aus mit dem Befehl **su** und Eingabe des Root-Passwortes geschehen.

oder Gruppen bestimmte Programme mit welchen Rechten ausführen dürfen. Um Kollisionen beim Editieren der Datei zu vermeiden, kann diese nicht mit einem herkömmlichen Editor bearbeitet werden. Stattdessen wird das Programm `visudo` verwendet:

```
root@linuxbox:~$ visudo
```

Die Bedienung entspricht jener des vi-Editors (vgl. Abschnitt A.2.2). Soll ein Benutzer `user1` mit `sudo` sämtliche Rechte erwerben können, `user2` dagegen nur den Paketmanager benutzen dürfen, kann dies durch Hinzufügen der folgenden Zeilen eingestellt werden:

```
user1 ALL=(ALL) ALL
user2 ALL=(root) /usr/bin/aptitude, /usr/bin/apt-get
```

An erster Stelle steht der Benutzer<sup>14</sup> selbst, an zweiter Stelle die Rechner, für die diese Einstellungen gelten. Da es sich hier um die Konfigurationsdatei für einen lokalen Rechner handelt, kann der Wert `ALL` gesetzt werden. Als dritter Wert folgt der Benutzer, unter dem der `sudo`-Befehl ausgeführt werden soll. Wird ein vom Eintrag in der ersten Spalte abweichender Benutzer angegeben, so werden dessen spezifischen Rechte verwendet. Zuletzt werden alle (`ALL`) oder bestimmte Befehle für diesen Eintrag freigeschaltet.

Wird die folgende Zeile einkommentiert, so sind für einen `sudo`-Aufruf keine Passwörter mehr erforderlich:

```
# %sudo ALL=NOPASSWD: ALL
```

Eine Anmerkung: Eine kurze Denkpause vor dem Ausführen eines Befehls mit Root-Rechten kann manchmal Wunder wirken. Daher sollte diese Zeile besser auskommentiert bleiben. Die Änderungen sind sofort nach dem Abspeichern der neuen Datei aktiv.

#### A.4.6 Cronjob-Verwaltung mit `crontab`

Für die Ausführung immer wiederkehrender Aufgaben steht unter Linux ein spezieller Dienst mit Namen *Cron* zur Verfügung (ausgeführt und überwacht vom Cron-Dämon). Mit diesem Dienst lassen sich Programme oder Skripte zu einer bestimmten Zeit ausführen, wobei der Rechner natürlich angeschaltet bleiben muss. Eine typische Anwendung ist die zeitliche Verschiebung von Aufgaben wie automatische Backups, Indizierung des Dateisystems, Sortieren oder Aufräumen von Verzeichnissen oder die Synchronisation mit anderen

<sup>14</sup> Für Gruppen wird das Zeichen `%` verwendet. So erteilt der Eintrag `%admin ALL=(ALL) ALL` allen Mitgliedern der Gruppe `admin` Ausführungsrechte ohne Einschränkungen.



Geräten. Diese Aufgaben können mit Cron-Jobs aus dem regulären Arbeitsablauf verschoben werden. Das Programm `crontab` dient der Verwaltung der Cron-Listen und kann mit folgenden Optionen aufgerufen werden:

- u <user> Ein Benutzer mit Root-Rechten kann mit dieser Option auf Cron-Tabellen eines Benutzers `user` zugreifen.
- e Editiert die aktuelle Cron-Tabelle und benutzt dazu den in der Umgebungsvariablen `$EDITOR` definierten Editor.
- l Zeigt die aktuelle Cron-Tabelle.
- r Entfernt die Cron-Tabelle des Benutzers.

Cron wird von Linux automatisch beim Systemstart aktiviert und überprüft jede Minute, ob neue Aktionen durchzuführen sind. Zusätzlich zu den individuellen Cron-Listen jedes Benutzers liegt in `/etc/crontab` zusätzlich eine zentrale Tabelle für alle Benutzer. Je nach verwendeter Distribution ist dort eine Verzeichnisstruktur für täglich, wöchentlich oder monatlich auszuführende Aufgaben enthalten. Treten während der Ausführung Probleme auf, so wird der zuständige Benutzer per `mail` informiert.

Die Tabelle selbst besteht aus sechs Spalten, wobei zunächst die Zeitangabe aufgeführt ist (Minute, Stunde, Tag, Monat, Wochentag), dann der auszuführende Befehl. Die einzelnen Spalten werden durch Leerzeichen oder Tabulatoren getrennt. Entsprechend ergibt sich die folgende Struktur:

*	*	*	*	*	auszuführender Befehl
-	-	-	-	-	
				+-----	Wochentag (0-6) (Sonntag = 0 oder 7)
			+-----		Monat (1-12)
		+-----			Tag (1-31)
	+-----				Stunde (0-23)
+-----					Minute (0-59)

Innerhalb der Felder können mehrere Einzelangaben stehen, die durch Kommata voneinander getrennt sind. Hierbei werden durch ein `/` Schrittwerten festgelegt: `*/3` bedeutet bspw., dass jeder dritte Wert berücksichtigt wird, `*` steht für *alle*. Da `cron` keine Kenntnis von der Variablen `$PATH` besitzt, sollte die Angabe der Befehle in Form absoluter Pfade erfolgen. Die folgende Tabelle zeigt exemplarisch die Einträge in einer Cron-Tabelle:

#M	S	T	M	W	Kommando
10	02	*	*	*	/home/user/cleanup.sh
*/5	*	*/2	*	*	/home/user/message.sh
59	23	*	*	0	cp /var/log/messages /log/backup/messages
*/10	7-12	*	*	1-5	/usr/local/bin/php /home/user/scripts/script.php
1	20	*	*	*	/home/user/scripts/sync.sh

Das erste Kommando wird jede Nacht um 02:10 Uhr ausgeführt. Die zweite gelistete Aufgabe alle fünf Minuten, jedoch nur an geraden Kalendertagen. Ein Kopieren der Log-Nachrichten erfolgt jeden Sonntag kurz vor 24:00 Uhr. Das

php-Skript wird vormittags zwischen 7 und 12 Uhr alle 10 Minuten aufgerufen, jedoch nur an Werktagen. Die Synchronisation erfolgt jeden Abend um 20:01 Uhr. Für eine einmalige Ausführung einer Aufgabe bietet sich weiterhin das Programm **at** an. Ist der Rechner zum gewünschten Ausführungszeitpunkt nicht aktiv, so wird der Job von **cron** beim nächsten Systemstart nicht nachgeholt, sondern verfällt. Sollte das Nachholen erwünscht sein, so ist **anacron** eine Alternative.

## A.5 Diagnose- und Failsafe-Modi

Im folgenden Text wird beschrieben, wie die vorgestellten Embedded-PCs jeweils in den Diagnose- bzw. Failsafe-Modus versetzt werden können. Die wichtigste Anwendung dieser Modi ist die Vorbereitung der Geräte auf die Übertragung einer neuen Firmware.

### A.5.1 Asus WL500g Premium

Um den Router in den Diagnosemodus zu versetzen, sind folgende Schritte auszuführen:

1. Stromkabel abziehen.
2. Hostrechner-IP im Bereich 192.168.1.2 bis 192.168.1.254 einstellen.
3. Den Anschluss LAN1 des Routers direkt mit dem PC verbinden.
4. Den schwarzen RESTORE-Knopf mit einem Stift gedrückt halten.
5. Das Stromkabel einstecken (der RESTORE-Knopf muss weiterhin gedrückt bleiben).
6. Wenn die Power-LED regelmäßig zu blinken beginnt, so befindet sich der Router im Diagnosemodus.
7. Der Router hat sich selbst die IP 192.168.1.1 zugewiesen und sollte auf **ping 192.168.1.1** von außen reagieren.
8. Der Router ist nun bereit zum Empfang einer Binärdatei mittels TFTP oder dem *Firmware Restoration Tool* von ASUS.

### A.5.2 Linksys WRT54G

Der Linksys WRT54G wird folgendermaßen in den Failsafe-Modus versetzt:

1. Stromkabel abziehen.

2. Hostrechner auf eine IP in 192.168.1.2 bis 192.168.1.254 einstellen.
3. Den Anschluss LAN1 des Routers direkt mit dem PC verbinden.
4. Falls eine neue Firmware aufgespielt werden soll, so sind nun in einer Shell folgende Kommandos auszuführen (sonst weiter mit Punkt 5.):

```
tftp 192.168.1.1
tftp> binary
tftp> trace
tftp> put openwrt-wrt54g-2.6-squashfs.bin
```

Jetzt das Stromkabel einstecken – der Download sollte sofort beginnen. Nach ca. zwei Minuten ist das Aufspielen beendet und es kann eine **telnet**-Verbindung aufgebaut werden (siehe Punkt 8).

5. Stromkabel einstecken und warten, bis die LED DMZ aufleuchtet.
6. Sobald die LED leuchtet, sofort mehrmals den Reset-Knopf drücken.
7. Die LED sollte nun mit einer Frequenz von ca. drei Hertz blinken.
8. Nun sollte ein **telnet** auf 192.168.1.1 möglich sein – Benutzername oder Passwort werden hierbei nicht benötigt.

Mit dem Befehl **passwd** kann nun auch ein neues Passwort gesetzt werden, falls dies der Grund für den Failsafe-Modus war. Falls das Flag **boot\_wait** nicht gesetzt ist, so sollte dies umgehend nachgeholt werden. Gleiches gilt auch nach dem Aufspielen einer neuen Firmware.

```
nvramp set boot_wait=on
nvramp set boot_time=10
nvramp commit && reboot
```

### A.5.3 Linksys NSLU2

Um die NSLU2 in den Upgrade-Modus zu versetzen, ist wie folgt zu verfahren:

1. NSLU2 abschalten.
2. Reset-Knopf mit einer Büroklammer drücken und gedrückt halten.
3. NSLU2 anschalten – die Ready/Status-LED sollte nun gelb leuchten.
4. Sobald die LED-Farbe auf orange/rot wechselt, den Reset-Knopf sofort loslassen – die Zeitspanne dafür ist mit ca. 1 s sehr kurz. Bei Erfolg blinkt die LED nun im Upgrade-Modus mit wechselnder Farbe (gelb, orange/rot), ansonsten muss ein neuer Versuch unternommen werden.
5. Die NSLU2 ist jetzt bereit für den Empfang eines neuen Images und sollte von **upslug2** erkannt werden.

# B

---

## Alternative Hardware-Plattformen

### B.1 Einführung

Im vorangegangenen Text wurden bereits verschiedene Hardware-Plattformen wie NSLU2, MicroClient, Asus und Alekto vorgestellt. Es existiert aber noch eine Vielzahl anderer Systeme, auf denen Linux zum Einsatz kommen kann. Im folgenden Abschnitt werden weitere interessante Plattformen vorgestellt, wobei die Liste keinen Anspruch auf Vollständigkeit erhebt.

Oft stellt im industriellen Umfeld der Formfaktor ein erstes Auswahlkriterium dar. Hierzu sei auf die Grafik 1.1 im Grundlagenkapitel verwiesen.

### B.2 Router

Einen Überblick über Router, auf denen OpenWRT problemlos läuft, bietet die Tabelle unter folgender Adresse:

<http://wiki.openwrt.org/TableOfHardware>

### B.3 Network Attached Storage

Neben der bekannten NSLU2 führen die folgenden Websites auch NAS-Alternativen zum Betrieb unter Debian auf:

<http://www.nslu2-linux.org> (Menüpunkt: Other Hardware)

<http://forum.nas-portal.org>

[http://nas-central.org/ALL\\_COMMUNITIES/Collection\\_of\\_NAS-Hacking\\_communities.html](http://nas-central.org/ALL_COMMUNITIES/Collection_of_NAS-Hacking_communities.html)

## B.4 Industrielle Kompaktsysteme

Neben den vorgestellten Alekto- und MicroClient-Systemen sind auch die Plattformen der folgenden Hersteller interessant:

<http://www.spectra.de>  
<http://www.abeco.de>  
<http://www.delta-components.com>  
<http://www.digitallogic.ch>

## B.5 Einplatinencomputer

Einplatinen- oder Single-Board-Computer (SBC) wurden bisher nur am Rande erwähnt. Die folgenden Adressen liefern weiterführende Informationen:

[http://www.linux-automation.de/cpu\\_boards/index\\_de.html](http://www.linux-automation.de/cpu_boards/index_de.html)  
<http://www.alix-board.de>  
<http://www.spectra.de>  
<http://www.congatec.de>  
<http://www.ipc-markt.de/>  
<http://www.delta-components.com>  
<http://www.digitallogic.ch>

Anmerkung: Besonders interessant für Embedded-Anwendungen sind aktuell die Low-Power- und Low-Cost-Prozessorfamilien Intel Atom und VIA Nano. Als Beispiel sei ein mittlerweile relativ bekanntes Board im Mini-ITX-Format angeführt: Das Intel D945GCLF-Board mit Atom-230-CPU und 945GC-Chipsatz (Preis ca. 50 EUR).

## B.6 Sonderlösungen

Abschließend werden in diesem Abschnitt noch einige Sonderlösungen vorgestellt, welche eine besonders kleine Bauform aufweisen, über exotische Peripherie verfügen oder speziell für eine bestimmte Aufgabe zugeschnitten sind.

So handelt es sich bei Smart Phones genau genommen nicht um eingebettete Systeme. Da sie aber viele Eigenschaften eingebetteter Systeme aufweisen und da auch der Software-Entwicklungsprozess identisch ist, sollen hier die wichtigsten Weblinks zu dem Thema aufgeführt werden:

<http://wiki.openmoko.org>  
<http://lipsforum.org/>

<http://android-developers.blogspot.com>  
<http://www.openhandsetalliance.com>  
<http://www.gnome.org/mobile>  
<http://www.maemo.org>  
<http://www.moblin.org>  
<http://www.linuxdevices.com/articles/AT9423084269.html>  
<http://trolltech.com/products/qtopia/qtopia-product-family/qtopia-phone-edition>

Für andere Anwendungen ist wiederum ein besonders kleiner Formfaktor interessant. Folgende linuxfähige Systeme zeichnen sich durch eine geringe Baugröße aus:

<http://www.kwikbyte.com/KB9260.html>  
<http://www.trenz-electronic.de/de/produkte/arm-cpu-module>  
<http://www.congatec.de/clx.html?&L=1>  
<http://www.via.com.tw/en/products/embedded/artigo>  
<http://de.kontron.com/products/computeronmodules/xboard>  
<http://www.fit-pc.com>

Auch Smart Cameras, also Kameras für den industriellen Einsatz mit einer eigenen Verarbeitungseinheit an Bord, werden mittlerweile von mehreren Herstellern angeboten. Die Linux-Derivate sind allerdings in diesem Bereich teilweise exotisch oder die Betriebssysteme sind nur an Linux angelehnt:

<http://www.scs-vision.ch/de/leanxcam/index.html>  
<http://www.vision-comp.com>  
<http://www.matrix-vision.com>  
<http://www.smartcamera.it/links.htm>  
<http://www.videor.com/default/3/0/0/0/imaging.html>  
<http://www.maxxvision.com/de/products/SmartKameras/SonySmartKameras/index.html>

# C

---

## Die IVT-Bibliothek

*Pedram Azad, Lars Pätzold*

### C.1 Einführung

Das *Integrating Vision Toolkit* (IVT) ist eine am Lehrstuhl Prof. Dillmann entstandene Bildverarbeitungsbibliothek. Sie ist unter der GNU-Lizenz frei im Quellcode erhältlich und kann auf Sourceforge<sup>1</sup> heruntergeladen werden. Detaillierte Anweisungen für die Installation der IVT finden sich im letzten Abschnitt dieses Kapitels.

Das vorliegende Kapitel beschreibt die einzelnen Schritte bis hin zur Komplettinstallation mit Anbindung der OpenCV-, OpenGL- und Qt-Bibliothek. Dies ermöglicht ein komfortables Prototyping auf dem Entwicklungs-PC. Für die Target-Plattform wird man im Regelfall allerdings nur die schlanke IVT-Bibliothek verwenden – die Anbindungen an die anderen genannten Bibliotheken sind optional. Der Bildeinzug über Video-for-Linux (V4L) kann über die Funktionalität der OpenCV oder auch auf Basis reiner IVT-Funktionen erfolgen (vgl. hierzu auch Kapitel 14).

Oberstes Ziel bei der Entwicklung der IVT war es, eine saubere, objektorientierte Architektur zugrunde zu legen und gleichzeitig die Algorithmen effizient umzusetzen. Ein Kernbestandteil, durch den sich die IVT von den meisten Bildverarbeitungsbibliotheken abhebt, so auch von der populären OpenCV, ist die Abstrahierung des Bildeinzugs durch eine entsprechende Kameraschnittstelle. Dadurch ist es möglich, Bildverarbeitungslösungen zu entwickeln, die aus Software-Sicht vollkommen unabhängig von der verwendeten Bildquelle sind. So reicht der Austausch einer Zeile für die Wahl des Kameramoduls aus, um ein Programm mit einer anderen Kamera verwenden zu können.

---

<sup>1</sup> <http://ivt.sourceforge.net>.

Bei der Implementierung lag das Hauptaugenmerk darauf, Abhängigkeiten zu Bibliotheken und zwischen Dateien der IVT untereinander zu vermeiden, wo immer möglich. Durch die strikte Trennung der Dateien, welche reine Eigenentwicklungen enthalten, von solchen, die Aufrufe an externe Bibliotheken kapseln, ist es auf einfache Weise möglich, die IVT nach Belieben zu konfigurieren und auch zu portieren. Der Kern der IVT ist in reinem ANSI C/C++ geschrieben und lässt sich ohne jegliche zusätzliche Bibliothek übersetzen. Wahlweise ist es möglich, Klassen bzw. Namensräume, welche Aufrufe an die Bibliotheken Qt, OpenCV und OpenGL kapseln, einzubeziehen. Dabei existieren keine Querabhängigkeiten, sodass diese Bibliotheken unabhängig voneinander hingenommen werden können.

## C.2 Architektur

In diesem Abschnitt wird eine kompakte Zusammenfassung der Architektur der IVT gegeben. Hierzu werden die Schnittstellen zu den Bildquellen, den grafischen Benutzeroberflächen, zu OpenCV und OpenGL erläutert. Am Ende dieses Kapitels veranschaulichen kurze Beispielprogramme die Verwendung der einzelnen Schnittstellen.

### C.2.1 Die Klasse `CByteImage`

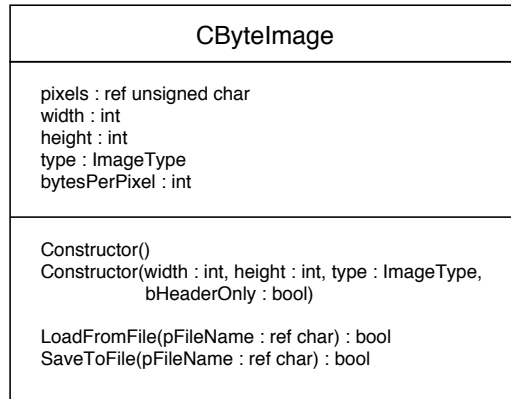
Den Kern der IVT bildet die Klasse `CByteImage`, welche in der Lage ist, ein 8 Bit-Graustufenbild und ein 24 Bit-Farbbild zu repräsentieren. Sie ist in reinem ANSI C++ geschrieben und somit plattformunabhängig. Zusätzlich zur reinen Repräsentation eines Bildes ist diese Klasse in der Lage, Bitmap-Dateien<sup>2</sup> zu lesen und zu schreiben.

Die public-Attribute dieser Klasse sind die ganzzahligen Variablen `width` und `height`, welche die Breite und Höhe des Bildes in Pixeln beschreiben, der Zeiger `pixels` des Typs `unsigned char*`, welcher auf den Anfang des Speicherbereichs des Bildes zeigt und die Variable `type`, welche den Wert `eGrayScale` für ein Graustufenbild und den Wert `eRGB24` für ein 24 Bit-Farbbild enthält. Bilder des Typs `eRGB24` können ebenfalls ein 24 Bit-HSV-Farbbild enthalten, da ein solches aus Sicht der Hinterlegung im Speicher identisch ist. Als zusätzliches Attribut wird die Variable `bytesPerPixel` zur Verfügung gestellt, welche den Wert Eins für Graustufenbilder und den Wert Drei für Farbbilder enthält.

---

<sup>2</sup> Bilddateien mit `.bmp` als Dateierweiterung.

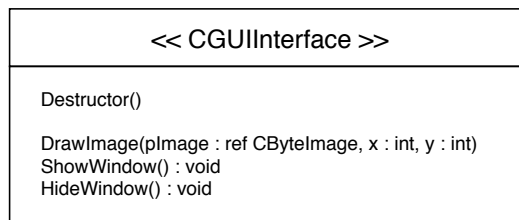




**Abb. C.1.** Darstellung der public-Attribute und Methoden der Klasse **CByteImage** in UML.

### C.2.2 Anbindung von grafischen Benutzeroberflächen

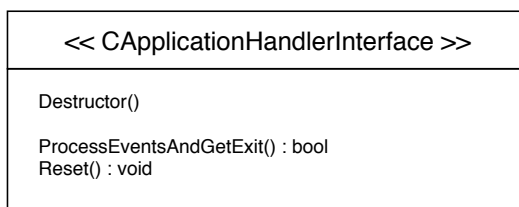
Die Elemente grafischer Benutzeroberflächen (engl. *Graphical User Interface* oder kurz GUI) für Bildverarbeitungsapplikationen lassen sich in zwei Gruppen unterteilen: Eingabeelemente wie Eingabefelder, Schieberegler, Buttons, etc. und die Anzeige von Bildern. Letztere Funktionalität ist in der IVT durch die Schnittstelle **CGUIInterface** gekapselt. Die Implementierung der Eingabeelemente bleibt der Applikation selbst überlassen und kann direkt mit der verwendeten Bibliothek vorgenommen werden.



**Abb. C.2.** Darstellung der Methoden der Schnittstelle **CGUIInterface** in UML.

Eine von **CGUIInterface** erbbende Klasse muss deren drei virtuelle Methoden und den virtuellen Destruktor implementieren. Die Methode **DrawImage** besitzt als Eingabe den Parameter **pImage** des Typs **CByteImage\*** für das darzustellende Bild und die zwei optionalen ganzzahligen Parameter **x** und **y**, mit denen es möglich ist, einen Offset vom linken oberen Eckpunkt des Fensters für den Beginn des Bildes anzugeben. Die Methoden **Show** und **Hide** sind für das Ein- bzw. Ausblenden des Fensters verantwortlich.

Zusätzlich muss für die meisten Bibliotheken eine Initialisierung zu Beginn des Programms durchgeführt werden und in jedem Durchlauf der Hauptschleife die Kontrolle für kurze Zeit abgegeben werden, um der Bibliothek die Möglichkeit zu geben, Ein- und Ausgaben zu behandeln. Die Kapselung dieser Aufrufe wird durch die Schnittstelle **CApplicationHandlerInterface** vorgenommen, welche aus den beiden virtuellen Methoden **Reset** und **ProcessEventsAndGetExit** besteht. Die Methode **Reset** muss zuerst aufgerufen werden, vor dem Anlegen und Anzeigen von Fenstern. Die Methode **ProcessEventsAndGetExit** sollte am Ende jedes Schleifendurchlaufes aufgerufen werden. Der Rückgabewert **true** signalisiert den Wunsch des Anwenders, die Applikation zu beenden.

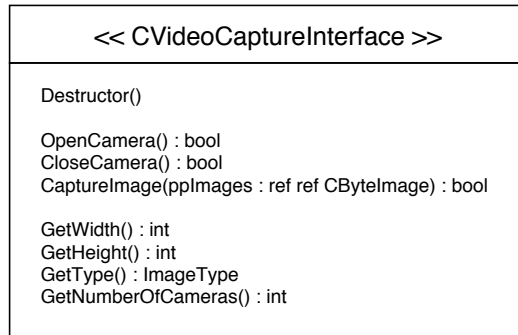


**Abb. C.3.** Darstellung der Methoden der Schnittstelle **CApplicationHandlerInterface** in UML.

In der IVT stehen als fertige Implementierungen dieser Schnittstelle die Klassen **COpenCVWindow/COpenCVApplicationHandler** und **CQTWindow/CQTApplicationHandler** zur Verfügung. In den Beispielapplikationen wird durchgängig die Qt-Implementierung verwendet, da es mit Qt auf einfache Weise möglich ist, Eingabeelemente hinzuzufügen. Die Implementierung verwendet die Qt-Bibliothek, welche für rein private oder wissenschaftliche Zwecke frei erhältlich ist. Detaillierte Anweisungen für die Installation von Qt finden sich im Abschnitt D dieses Anhangs. Ein Beispiel für die Verwendung dieser Schnittstelle mit Qt wird in Abschnitt C.3.2 gegeben.

### C.2.3 Anbindung von Bildquellen

In der IVT wird für jede Bildquelle ein Modul implementiert, welches über die fest vorgegebene Schnittstelle **CVideoCaptureInterface** Bilder des Typs **CByteImage** liefert. Die Schnittstelle ist derart definiert, dass eine beliebige Anzahl von Bildern bei einem Aufruf übertragen werden kann, wie dies bei Mehrkamerasystemen erforderlich ist. Im Folgenden wird die Bezeichnung *Kameramodul* synonym für das Modul einer Bildquelle verwendet.



**Abb. C.4.** Darstellung der Methoden der Schnittstelle CVideoCaptureInterface in UML.

Ein Kameramodul muss die sieben virtuellen Methoden der Schnittstelle und den virtuellen Destruktor implementieren. Die Methode **OpenCamera** führt die notwendigen Initialisierungen durch, startet die Bildaufnahme und liefert einen booleschen Rückgabewert, welcher den Erfolg der Initialisierung wiedergibt. Die notwendigen Parameter für die Konfiguration des Moduls, wie beispielsweise die Wahl der Auflösung oder der Kodierung, unterscheiden sich von Bildquelle zu Bildquelle und sind daher über den Konstruktor des jeweiligen Moduls zu wählen. Die Methode **CloseCamera** beendet die Bildaufnahme und löscht eventuell verwendete Objekte und Speicherbereiche. Die Methode **CaptureImage** besitzt als Eingabe den Parameter **ppImages** des Typs **CByteImage\*\***. Dadurch ist es möglich, beliebig viele Bilder mit einem Aufruf zu übertragen. Die Instanzen von **CByteImage** müssen bereits angelegt sein und in Größe, Typ und Anzahl mit den Anforderungen des Kameramoduls übereinstimmen. Zu diesem Zweck kann diese Information über die Methoden **GetWidth**, **GetHeight**, **GetType** und **GetNumberOfCameras** abgerufen werden. Um gültige Werte zu erhalten, muss zuvor die Methode **OpenCamera** erfolgreich aufgerufen worden sein. Ein Beispiel für die Verwendung eines Kameramoduls wird in Abschnitt C.3.3 gegeben.

#### C.2.4 Anbindung der OpenCV

Die OpenCV<sup>3</sup> ist eine populäre und oft verwendete Bildverarbeitungsbibliothek, welche ebenfalls auf Sourceforge im Quellcode zur Verfügung gestellt wird. Sie bietet ein breites Spektrum an Bildverarbeitungsalgorithmen, verfügt jedoch über keine objektorientierte Gesamtarchitektur.

Aufgrund der Mächtigkeit der OpenCV ist diese optional in die IVT eingebunden. Hierzu wurden Funktionen der OpenCV gekapselt und können über Auf-

<sup>3</sup> <http://sourceforge.net/projects/opencvlibrary>.

rufe an die IVT, welche meist auf Instanzen der Klasse `CByteImage` operieren, verwendet werden. Auf diese Weise erfolgt die Verwendung von Funktionen aus der OpenCV vollkommen transparent, d. h. die aufgerufenen Methoden sind unabhängig von der OpenCV. Da die Bilder selbst nicht konvertiert werden müssen, sondern lediglich der aus wenigen Bytes bestehende Header eines Bildes umgewandelt werden muss, erfolgen die Aufrufe an die OpenCV praktisch ohne Mehraufwand. Dateien, welche Funktionen der OpenCV kapseln, tragen als Endung die Buchstaben `CV`, wie beispielsweise `ImageProcessorCV.h` und `ImageProcessorCV.cpp`. Ein Beispiel für die Verwendung der OpenCV innerhalb der IVT wird in Abschnitt C.3.4 gegeben.

### C.2.5 Anbindung von OpenGL über Qt

OpenGL<sup>4</sup> ist eine Spezifikation einer plattformunabhängigen Programmierschnittstelle zur Entwicklung von 3D-Grafikanwendungen. Eine Implementierung dieser Schnittstelle steht auf allen gängigen Betriebssystemen zur Verfügung, so auch unter Windows, Mac OS und Linux.

Für manche 3D-Bildverarbeitungsapplikationen kann es nützlich sein, das Ergebnis ebenfalls in 3D zu visualisieren. Zu diesem Zweck kapselt die IVT mit der Klasse `COpenGLVisualizer` die 3D-Primitive Kugel und Zylinder, welche über die Methoden `DrawSphere` und `DrawCylinder` gezeichnet werden können. Bevor diese Methoden verwendet werden können, muss die Methode `Init` aufgerufen werden, welche als Eingabeparameter die Breite und die Höhe des Grafikbereiches erwartet. Soll der Inhalt des Grafikbereiches gelöscht werden, so muss die Methode `Clear` aufgerufen werden. Nachfolgend wird beschrieben, wie der Grafikbereich in einem Fenster dargestellt werden kann. Als Alternative ist es ebenfalls möglich, den Grafikbereich direkt in ein RGB24-Farbbild des Typs `CByteImage` zu schreiben. Hierzu muss ein Bild der entsprechenden Größe – wie in der Methode `Init` zuvor angegeben – angelegt und an die statische Methode `GetBuffer` übergeben werden. Da es sich um eine statische Methode handelt, kann diese direkt mit `COpenGLVisualizer::GetBuffer` aufgerufen werden.

Mithilfe der Klasse `CQTGLWindow` ist es möglich, den OpenGL-Grafikbereich in einem Fenster darzustellen. Im Konstruktor muss die Größe des Fensters angegeben werden. Diese sollte mit den in der Methode `Init` aus `COpenGLVisualizer` angegebenen Parametern übereinstimmen, da sonst der darzustellende Bereich abgeschnitten wird bzw. nur einen Teil des Fensters ausfüllt. Um den Inhalt des Fensters zu aktualisieren, muss die Methode `Update` aufgerufen werden. Ein Beispiel für die Verwendung von OpenGL innerhalb der IVT wird in Abschnitt C.3.5 gegeben.

---

<sup>4</sup> Open Graphics Library.

## C.3 Beispielapplikationen

In diesem Abschnitt werden zu den erläuterten Schnittstellen und angebundenen Bibliotheken Beispiele gegeben. Jedes Beispiel besteht aus einer kurzen Beschreibung und einer Implementierung in C++. Die Quelltexte hierzu finden sich am Ende des Kapitels.

### C.3.1 Verwendung der Basisfunktionalität

Diese Applikation soll als einfachstes Beispiel für die Verwendung der IVT dienen. Es wird ein Bild geöffnet, mithilfe der Funktion `ConvertImage` aus `ImageProcessor` in ein Graustufenbild konvertiert und anschließend ein Gradientenfilter mit `CalculateGradientImageSobel`, ebenfalls aus `ImageProcessor`, angewendet. Das Ergebnis wird in eine Bitmap-Datei geschrieben. Für das Übersetzen und Aufrufen dieser Applikation sind keine weiteren Bibliotheken notwendig. Der Name dieser Applikation lautet `simpleapp`.

### C.3.2 Verwendung einer grafischen Benutzeroberfläche

Diese Applikation zeigt die Verwendung der Schnittstelle `CGUIInterface`. Zunächst wird ein Bild geöffnet. Bei Erfolg wird anschließend ein Qt-Fenster angelegt und angezeigt, um dann in einer Schleife das geladene Bild darzustellen. Für das Übersetzen und Aufrufen dieser Applikation ist die Qt-Bibliothek erforderlich. Detaillierte Anweisungen für die Installation von Qt finden sich im Abschnitt D dieses Anhangs. Der Name dieser Applikation lautet `guiapp`.

Eine Anmerkung: Parallel zur Drucklegung dieses Buches erfährt die IVT einen Versionssprung. Danach wird die IVT problemlos mit Qt3.xx, mit Qt4.xx und auch ohne Qt betriebsfähig sein. Der aktuelle Stand, Details zur Installation und nähere Informationen zu einer evtl. notwendigen Anpassung der Makefiles sind unter folgenden URL zu erfahren:

<http://ivt.sourceforge.net/installation.html>

<http://www.praxisbuch.net> (dort: *Computer Vision* bzw. *Embedded Linux*).

### C.3.3 Verwendung eines Kameramoduls

In dieser Applikation wird die Verwendung der in Abschnitt C.2.3 erläuterten Schnittstelle zu Bildquellen gezeigt. Um auch Lesern, die keine Kamera besitzen, das Übersetzen und Ausführen dieser Applikation zu ermöglichen, wird als Kameramodul die Klasse `CBitmapCapture` verwendet. Diese ist in der Lage,

eine Bilddatei von einem anzugebenden Pfad zu lesen und eine Kamera mit einem statischen Bild zu simulieren. Zu Beginn wird dieses Modul angelegt und initialisiert. Bei Erfolg wird ein Bild entsprechender Größe und entsprechenden Typs angelegt. Anschließend wird in jedem Durchlauf der Hauptschleife das Bild an das Kameramodul für die Übertragung des aktuellen Bildes übergeben und dann in einem Fenster angezeigt.

Soll ein anderes Kameramodul verwendet werden, um beispielsweise eine reale Kamera anzubinden, so muss lediglich die Zeile für das Anlegen der Instanz des Kameramoduls zu Beginn der `main`-Routine modifiziert werden.

### C.3.4 Verwendung der OpenCV

Diese Applikation zeigt die Verwendung der OpenCV innerhalb der IVT über die zur Verfügung stehenden Kapselungen. In diesem Beispiel wird als Alternative für das Öffnen eines Bildes mithilfe der Methode `LoadFromFile` der Klasse `CByteImage` die Funktion mit demselben Namen aus `ImageProcessorCV` verwendet. Diese kapselt einen Aufruf an die Funktion `cvLoadImage` der OpenCV und ist in der Lage, auch andere Bildformate einzulesen, wie JPG, PPM, TIFF und PNG. Um diese Funktion verwenden zu können, ist die Teilbibliothek `highgui` der OpenCV notwendig.

Zu Beginn der Applikation wird ein Bild geöffnet und bei Erfolg in ein Graustufenbild umgewandelt. Anschließend wird das Ergebnis mithilfe der Funktion `Resize` aus `ImageProcessorCV` in ein Bild halber Breite und Höhe konvertiert. Auf das Ergebnis wird ein Gradientenfilter angewendet und das Endergebnis in eine Bitmap-Datei gespeichert, wie bereits in der Applikation aus Abschnitt C.3.1. Wie zu sehen ist, können gekapselte Aufrufe an die OpenCV und reine IVT-Aufrufe beliebig ausgetauscht und miteinander vermischt werden.

### C.3.5 Verwendung der OpenGL-Schnittstelle

In dieser Applikation wird die Drehung eines einfachen 3D-Objektes, bestehend aus zwei Kugeln und einem Zylinder animiert. Hierzu wird zunächst Qt initialisiert und ein Fenster des Typs `CQTGLWindow` angelegt. Anschließend wird die OpenGL-Kapselung erzeugt und initialisiert. Mithilfe einer 2D-Rotation werden die Endpunkte `point1` und `point2` in der Ebene  $y = 0$  in jedem Durchlauf der Hauptschleife um den aktuellen Winkel `angle` gedreht.

Um auf unterschiedlichen Rechnern dieselbe Bildfrequenz (engl. *frame rate*) zu erhalten, wird mithilfe eines Timers diese auf 30 Hz festgelegt. Hierzu liefert die Funktion `get_timer_value` einen ganzzahligen Wert des Typs `unsigned int` zurück, welcher einen relativen Zeitwert in Mikrosekunden beschreibt. Wird als Argument der Parameter `true` an `get_timer_value` übergeben, so wird der Timer auf Null zurückgesetzt.

## C.4 Übersicht zu weiterer Funktionalität der IVT

In den vorherigen Abschnitten wurden grundlegende Klassen und Schnittstellen der IVT vorgestellt, für die Repräsentation und die Visualisierung von Bildern sowie für die Anbindung nützlicher Bibliotheken. In diesem Abschnitt soll nun eine kurze Übersicht über die wichtigsten von der IVT zur Verfügung gestellten Routinen gegeben werden. Die Angabe von Verzeichnissen ist jeweils als Unterverzeichnis von `IVT/src` zu verstehen. Handelt es sich um Klassen, so tragen diese den zusätzlichen Buchstaben „C“ im Vergleich zu den Dateinamen. Bei Namensräumen sind die Namen identisch.

### Kamera-Kalibrierung (Verzeichnis `Calibration`)

<code>C Calibration:</code>	Kameraabbildungsfunktionen für eine einzelne Kamera
<code>CRectification(CV):</code>	Durchführung einer Rektifizierung
<code>CStereoCalibration:</code>	Berechnungen für Stereokamerasysteme
<code>StereoCalibrationCV:</code>	Berechnung der Rektifizierungsparameter für eine gegebene Instanz von <code>CStereoCalibration</code>
<code>CUndistortionCV:</code>	Durchführung einer Entzerrung

### Operationen auf Bildern und Repräsentationen (Verzeichnis `Image`)

<code>CByteImage:</code>	Datenstruktur für die Repräsentation von 8 Bit-Graustufenbildern und RGB24-Farbbildern
<code>CShortImage:</code>	Datenstruktur für die Repräsentation von 16 Bit-Graustufenbildern
<code>ImageAccessCV:</code>	Laden und Speichern von Bildern mit der OpenCV
<code>ImageProcessor(CV):</code>	Punktoperatoren, Filter, morphologische Operatoren, Konvertierungsroutinen, etc.
<code>IplImageAdaptor:</code>	Konvertierung zwischen <code>CByteImage</code> (IVT) und <code>IplImage</code> (OpenCV)
<code>PrimitivesDrawer(CV):</code>	Zeichnen von 2D-Primitiven wie Kreise, Ellipsen, Rechtecke, Linien, etc.
<code>StereoVision(SVS):</code>	Berechnung von Tiefenkarten

**Mathematische Routinen** (Verzeichnis Math)

<b>CFloatMatrix:</b>	Datenstruktur für die Repräsentation einer Matrix von Werten des Datentyps <code>float</code> (kompatibel zu <code>CByteImage</code> und <code>CShortImage</code> )
<b>CMatd:</b>	Datenstruktur und Operationen für das Rechnen mit Matrizen beliebiger Dimension
<b>CVecd:</b>	Datenstruktur und Operationen für das Rechnen mit Vektoren beliebiger Dimension
<b>Math2d:</b>	Datenstrukturen und Operationen für effizientes Rechnen mit Vektoren und Matrizen in 2D
<b>Math3d:</b>	Datenstrukturen und Operationen für effizientes Rechnen mit Vektoren und Matrizen in 3D

**C.5 Installation**

Die vorliegende Anleitung beschreibt die Einrichtung einer lauffähigen Umgebung für die Programmierung mit der IVT-Bibliothek unter Linux.<sup>5</sup> Dazu gehört auch die Einrichtung der von der IVT unterstützten Bibliotheken OpenCV und Qt sowie eines Treibers für 1394-Firewire-Kameras. Details zu den (gängigeren) USB-Webcams finden sich in Kapitel 14.

Um möglichst wenige Vorkenntnisse vorauszusetzen, wurde die Anleitung in vielen Teilen bewusst ausführlich gehalten. Damit soll der Einstieg in die Verwendung der IVT erleichtert werden. Tatsächlich sind relativ viele Schritte notwendig, bis man sich der Programmierung widmen kann. Trotz des Mehraufwandes wird empfohlen, zumindest auf dem Entwicklungs-PC, alle aufgeführten Bibliotheken zu installieren. Ist die Verwendung einer 1394-Kamera nicht vorgesehen, dann können die entsprechenden Schritte, in denen in irgendeiner Weise „1394“ vorkommt, übersprungen werden.

Es ist zu beachten, dass die hier vorgestellte Software lizenzgemäß nur für private und/oder wissenschaftliche Zwecke verwendet werden darf, nicht jedoch für den gewerblichen Einsatz. Für genauere Informationen zu den Lizenzbestimmungen sei auf die zu jeder Software angegebene Internetseite verwiesen.

---

<sup>5</sup> Diese Anleitung liegt unter <http://ivt.sourceforge.net/installation.html> auch aktualisiert und gepflegt als Online-Version vor.



### C.5.1 OpenCV

#### a.) Download der OpenCV

Unter <http://sourceforge.net/projects/opencvlibrary> im Download-Bereich unter dem Package `opencv-linux` folgt man dem weiterführenden Link *1.0*. Dort lässt sich die Datei `opencv-1.0.0.tar.gz` herunterladen.

#### b.) Entpacken

In einer Konsole wechselt man zum Entpacken in das Verzeichnis des heruntergeladenen Archivs und führt anschließend den folgenden Befehl aus:

```
tar xfvz opencv-1.0.0.tar.gz
```

#### c.) Erstellen der OpenCV-Bibliotheken

Nach dem Wechseln in das entpackte Verzeichnis `opencv-1.0.0` führt man die folgenden Befehle nacheinander aus:

```
./configure
make
make install
ldconfig
```

**Hinweis:** Für die Befehle `make install` und `ldconfig` sind Root-Rechte erforderlich.

Im Anschluss daran sollten sich im Verzeichnis `/usr/local/include/opencv` die Include-Dateien der OpenCV befinden.

### C.5.2 Qt

Für eine detaillierte Beschreibung zu Qt sei an dieser Stelle auf den Anhang D verwiesen. Dort wird auch die Installation erklärt. Anzumerken ist, dass die IVT parallel zur Drucklegung dieses Buches einen Versionssprung erfährt. Danach wird die IVT problemlos mit Qt3.xx, mit Qt4.xx und auch ohne Qt betriebsfähig sein. Der aktuelle Stand, Details zur Installation und nähere Informationen zu einer evtl. notwendigen Anpassung der Makefiles sind unter folgenden URL zu erfahren:

<http://ivt.sourceforge.net/installation.html>

<http://www.praxisbuch.net> (dort: *Computer Vision* bzw. *Embedded Linux*).

Ein weiterer Hinweis: Für MicroClient-/Puppy-Anwender, die direkt auf dem Embedded-Gerät entwickeln oder aus anderen Gründen Qt auch dort installieren möchten, empfiehlt sich die Verwendung einer vorcompilierten Version. Vgl. hierzu auch:

<http://www.puppylinux.org/wiki/multi-lingual-puppy/technical-how-tos/making-scim-work-with-qt-almost-beginner-friendly>

### C.5.3 Firewire und libdc1394/libraw1394

#### *a.) Installation von libdc1394/libraw1394*

Die beiden Bibliotheken libdc1394 und libraw1394 können ebenfalls mit einem Package-Tool installiert werden. Unter Debian Linux z. B. mittels:

```
apt-get install libdc1394
apt-get install libraw1394
```

#### *b.) Einrichten einer Firewire-Kamera (optional)*

Um die Kamera unter Linux zu verwenden, müssen die vier Kernelmodule namens raw1394, video1394, ohci1394 und ieee1394 aktiviert werden. Dies erfolgt unter Debian Linux mithilfe des Konsolenbefehls **modprobe**. Dabei genügen folgende Aufrufe, um alle vier Module zu aktivieren:

```
modprobe raw1394
modprobe video1394
```

Ob die vier Module vorhanden sind, lässt sich kontrollieren wie folgt:

```
lsmod | grep 1394
```

Dieser Befehl sollte die o. g. vier Module auflisten. Sollte dies nicht der Fall sein, kann versucht werden, jedes Modul einzeln mit dem Befehl **modprobe** hinzuzufügen. Damit eine Anwendung auf die Schnittstelle zur Kamera zugreifen kann, ist der Anwender in die entsprechenden Gruppen der Geräte unter **/dev/raw1394** und **/dev/video1394** einzutragen.

Für das Eintragen in die Gruppe – normalerweise handelt es sich um die Gruppe **video** – sind Root-Rechte erforderlich.

### C.5.4 IVT

#### a.) Download der IVT

Hierzu lädt man im Internet unter <http://sourceforge.net/projects/IVT/> im Download-Bereich das Archiv, z. B. `ivt-1.0.6.tar.gz`, herunter.

#### b.) Entpacken

In einer Konsole wechselt man zum Entpacken in das Verzeichnis des heruntergeladenen Archivs und führt folgenden Befehl aus:

```
tar xfvz ivt-1.0.6.tar.gz
```

Anschließend sollte sich im aktuellen Verzeichnis das Verzeichnis IVT befinden.

#### c.) Konfiguration

Die IVT bietet die Möglichkeit der Konfiguration der Bibliothek, bevor sie erstellt wird. Diese Konfiguration wird in der Datei `IVT/src/Makefile.base` vorgenommen. Dazu muss `Makefile.base` mit einem Text-Editor (z. B. geany) geöffnet und bearbeitet werden. Im Folgenden werden die wichtigsten Konfigurationsmöglichkeiten aufgelistet. Dabei handelt es sich um Konfigurationsvariablen, die entweder auf 1 oder auf 0 gesetzt werden können. Um bestimmte Teile der IVT-Bibliothek einzubinden oder auszuschließen, muss die entsprechende Konfigurationsvariable auf 1 für die Einbindung und auf 0 für den Ausschluss gesetzt werden.

`USE_QT = 1`

Der Umfang der IVT-Bibliothek wird um Klassen erweitert, die es ermöglichen, auf einfache Art und Weise grafische Benutzerschnittstellen mit Qt zu erstellen. Die zugehörigen Quelltext-Dateien mit den Dateinamen `QT*` befinden sich in `IVT/src/gui`.

`USE_QT_GUI = 0`

Dieser Schalter ist aktuell noch auf 0 zu setzen. Zu Details vgl. die aktualisierte Online-Dokumentation unter <http://ivt.sourceforge.net/installation.html>.

`USE_OPENCV = 1`

Ein Teil der IVT-Bibliothek greift auf die OpenCV-Bibliothek zurück. Diesen Teil bilden die Quelltext-Dateien mit den Endungen `CV.h` bzw. `CV.cpp`.

USE\_OPENGL = 1

OpenGL und GLU werden von der Klasse `COpenGLVisualizer` verwendet, um Kugeln und Zylinder darzustellen. Die Quelltext-Dateien für diese Klasse befinden sich unter `IVT/src/Visualizer`.

USE\_HIGHGUI = 1

HighGUI ist Bestandteil der OpenCV-Bibliothek. Als Alternative zu Qt lassen sich damit Fenster und Bilder grafisch darstellen. Die entsprechenden Quelltext-Dateien `OpenCV*`, die HighGUI verwenden, befinden sich in `IVT/src/gui`.

USE\_LIBDC1394 = 1

Eine Schnittstelle zur Bibliothek `libdc1394` ermöglicht die Ansteuerung von IEEE1394-Kameras (Firewire-Kameras). Die Quelltext-Dateien der Schnittstelle heißen `Linux1394Capture.*` und befinden sich in `IVT/src/VideoCapture`.

Neben diesen Variablen lassen sich im unteren Teil der Datei die Verzeichnispfade für Include- und Bibliotheksdateien sowie die Dateinamen der Bibliotheken ändern. Je nach Linux-Version bzw. je nach verwendeter Bibliotheksversion kann hier eine Anpassung notwendig werden (ein Beispiel: Qt3 → Qt4).

Für eventuelle Anpassungen folgt eine kurze Erklärung der entsprechenden Variablen:

INCPATHS\_BASE

Enthält alle Verzeichnispfade für Include-Dateien. Das Hinzufügen eines Pfades erfolgt mithilfe des Operators `+=` und des dem Pfad vorangestellten Parameters `-I`. Beispielsweise `INCPATHS_BASE += -I/usr/include/qt3`

LIBPATHS\_BASE

Enthält alle Verzeichnispfade für Bibliotheksdateien. Das Hinzufügen eines Pfades erfolgt mithilfe des Operators `+=` und des dem Pfad vorangestellten Parameters `-L`. Beispielsweise `LIBPATHS_BASE += -L/usr/lib/qt3/lib`

LIBS\_BASE

Enthält alle Dateinamen der Bibliotheken. Das Hinzufügen von Dateinamen erfolgt mithilfe des Operators `+=` und des den Dateinamen vorangestellten Parameters `-l`. Beispielsweise `LIBS_BASE += -lqt-mt -lvtgui`

#### *d.) Erstellung der IVT-Bibliotheken*

Die Bibliotheken werden erstellt, indem der Befehl `make` im Verzeichnis `IVT/src` ausgeführt wird. Vor jedem erneuten Ausführen von `make` sollte der Befehl `make clean` ausgeführt werden, um die IVT für eine vollständige Neukompilierung zurückzusetzen.

*e.) Beispielanwendungen*

Zur Überprüfung der Installation oder zum Einstieg in die Programmierung mit der IVT eignet sich die Beispielanwendung SimpleApp im Verzeichnis `IVT/examples/SimpleApp` im IVT-Verzeichnis. Nach dem Wechseln in das Verzeichnis `SimpleApp` wird mit dem Befehl `make` eine ausführbare Datei mit dem Namen `simpleapp` erstellt. Führt man diese mit `./simpleapp ../../files/dish_scene_left.bmp` aus, so wird bei korrekter Installation aller Bibliotheken eine Datei namens `output.bmp` im selben Verzeichnis erzeugt. Zusammengefasst lauten die Aufrufe:

```
make
simpleapp ./simpleapp ../../files/dish_scene_left.bmp
```

Hiermit ist die Einrichtung und damit auch die Anleitung abgeschlossen. Im Verzeichnis `IVT/examples` finden sich zahlreiche weitere Beispielanwendungen, mit denen sich die Möglichkeiten der IVT testen und erlernen lassen.

Feb 20, 07 22:48	Beispielapplikationen	Page 1/5
<pre> // ***** // Dateiname: simpleapp.cpp // Autor: Pedram Azad // ***** // ***** #include "Image/ImageProcessor.h" #include "Image/ByteImage.h" #include &lt;stdio.h&gt;  int main() {     CByteImage image;      // Bild laden     if (!image.LoadFromFile("../files/dish_scene_left.bmp"))     {         printf("Fehler: Konnte Bild nicht öffnen\n");         return 1;     }      // Farbbild in ein Graustufenbild konvertieren     CByteImage gray_image(image.width, image.height, CByteImage::eGrayScale);     ImageProcessor::ConvertImage(&amp;image, &amp;gray_image);      // Gradientenbild berechnen     ImageProcessor::CalculateGradientImagesobel(&amp;gray_image, &amp;gray_image);      // Ergebnisbild abspeichern     gray_image.SaveToFile("output.bmp");     printf("Ergebnisbild wurde in 'output.bmp' abgespeichert\n");      return 0; } </pre>		

Feb 20, 07 22:48	Beispielapplikationen	Page 2/5
<pre> // ***** // Dateiname: guiapp.opp // Autor: Pedram Azad // ***** // ***** #include "Image/ByteImage.h" #include "gui/QTApplicationHandler.h" #include "gui/QTWindow.h" #include &lt;stdio.h&gt;  int main(int argc, char **args) {     CByteImage image;      // Bild laden     if (!image.LoadFromFile("../files/dish_scene_left.bmp"))     {         printf("Fehler: Konnte Bild nicht öffnen\n");         return 1;     }      // QT initialisieren     CQTApplicationHandler qtApplicationHandler(argc, args);     qtApplicationHandler.Reset();      // Fenster anlegen und anzeigen     CQTWindow window(image.width, image.height);     window.Show();      // Hauptschleife     while (!qtApplicationHandler.ProcessEventsAndGetExit())     {         window.DrawImage(&amp;image);          return 0;     } } </pre>		



Feb 20, 07 22:48	Beispielapplikationen	Page 5/5
<pre> // ***** // Dateiname: openglapp.cpp // Autor:      Pedram Azad // Mail:       pedram.azad@uni-erlangen.de // *****  #include "gui/QTGLWindow.h" #include "gui/QTApplicationHandler.h" #include "Visualizer/OpenGLVisualizer.h" #include "Math/Math3d.h" #include "Math/Math2d.h" #include "Math/Math1d.h" #include "Helpers/helpers.h" #include &lt;math.h&gt;  int main(int argc, char **args) {     // QT initialisieren     QTApplicationHandler qtApplicationHandler(argc, args);     qtApplicationHandler.Reset();      // Fenster und OpenGL-Kapselung anlegen     OpenGLVisualizer visualizer;     COpenGLVisualizer visualizer;      // OpenGL initialisieren     visualizer.Init(640, 480);      const double r = 250;     double angle = 0;      // Hauptschleife     while (!qtApplicationHandler.ProcessEventsAndGetExit())     {         unsigned int t = get_timer_value(true);          const Vec3d p1 = { r * cos(angle), 0, r * sin(angle) + 1500. };         const Vec3d p2 = { r * cos(angle + PI), 0, r * sin(angle + PI) + 1500. };          // Grafik-Bereich löschen         visualizer.Clear();          // Kugeln und Zylinder zeichnen         visualizer.DrawSphere(p1, 100, COpenGLVisualizer::red);         visualizer.DrawSphere(p2, 100, COpenGLVisualizer::red);         visualizer.DrawCylinder(p1, p2, 50, 50, COpenGLVisualizer::blue);          // Fensterinhalt neu zeichnen         window.Update();          // Drehwinkel erhöhen         angle += 0.05;          // Timing         const double T = 1000000.0 / 30; // 30 fps (frames per second)         while (get_timer_value() &lt; t + T);     }      return 0; } </pre>		



# D

---

## Die Qt-Bibliothek

Das vorliegende Kapitel dient der Vermittlung der Grundlagen zur Qt-Bibliothek als Vorbereitung für Kapitel 13. Besonderes Augenmerk liegt auf der Verwendung des Programms *Qt-Designer* und dessen Erweiterungen. Die zugehörigen Quelltext-Beispiele befinden sich im Verzeichnis `<embedded-linux-dir>/examples/qt/`.

### D.1 Einführung

#### D.1.1 Installation und Grundlagen

Für die Installation unter Ubuntu wird die Open-Source-Edition von Qt für Linux/X11 benötigt. Diese ist in Form mehrerer Debian-Pakete verfügbar und wird mit folgendem Befehl installiert:

```
$ sudo apt-get install libqt4-dev qt4-designer qt4-doc qt4-dev-tools
```

Alternativ besteht die Möglichkeit, Qt von der Website des Herstellers Trolltech herunterzuladen und selbst zu übersetzen.<sup>1</sup> Dazu wird die aktuelle Version als .tar.gz-Datei geladen (momentan Version 4.4.3), und den Installationsanweisungen unter <http://doc.trolltech.com/4.4/installation.html> für Qt/X11 gefolgt. Neben weiteren Versionen für Windows und Mac OS X ist auch eine Variante *Qt for Embedded Linux* verfügbar. Allerdings ist die Liste der unterstützten Embedded-Geräte noch sehr überschaubar, sodass in diesem Kapitel lediglich die X11-Version zur Entwicklung von Host-Anwendungen auf einem PC vorgestellt wird.

Im zweiten Fall muss das Qt-Verzeichnis zur Umgebungsvariablen `PATH` hinzugefügt werden, um Werkzeuge wie `qmake` oder `moc` an der Kommandozeile

---

<sup>1</sup> Vgl. <http://www.trolltech.com/developer/downloads>.

verfügbar zu machen. Dazu sind folgende Zeilen der Datei `~/.bashrc` hinzuzufügen:<sup>2</sup>

```
PATH=/usr/local/Trolltech/Qt-4.4.3/bin:$PATH
export PATH
```

Abschnitt A.4.2 erläutert die verschiedenen Möglichkeiten zur Belegung der Umgebungsvariablen. Ein Aufruf von `qmake -v` in der Kommandozeile sollte dann folgendes Ergebnis liefern:

```
$ qmake -v
QMake version 2.01a
Using Qt version 4.4.3 in /usr/lib
```

## Qt sagt „*Hello World*“

Nachdem nun die Grundlagen geschaffen sind, kann mit der ersten Qt-Anwendung begonnen werden. Grundsätzlich ist zu sagen, dass mit dem Qt-Tutorial eine exzellente Möglichkeit für einen ersten Einstieg zur Verfügung steht.<sup>3</sup> An dieser Stelle sollen deshalb lediglich einige elementare Dinge, wie die Qt-Projektorganisation und das Prinzip der *Signals und Slots* erklärt werden. Eine einfache Qt-Anwendung kann wie folgt aussehen (vgl. Beispiel `qt.basic.hello`):

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton hello("Hello world!");
    hello.resize(200, 50);

    hello.show();
    return app.exec();
}
```

In jeder Anwendung muss genau ein Objekt vom Typ `QApplication` vorhanden sein, das als Schnittstelle zwischen der Qt-Anwendung und der Benutzeroberfläche dient. Ein weiteres Objekt vom Typ `QPushButton` wird später einen Knopf in der Anzeige darstellen. Für diese beiden Elemente sind entsprechende Header-Dateien einzubinden. Anzeigeelemente sind in Qt zunächst nicht sichtbar und erscheinen erst nach einem Aufruf von `show()`. Mit dem Befehl `app.exec()` übergibt die `main()`-Funktion die Kontrolle an die Qt-Anwendung und startet diese. Der Rückgabewert wird nach Beendigung durchgereicht.

<sup>2</sup> Das Qt-Verzeichnis kann abweichen und ist ggf. über `sudo find / -name "qmake"` zu ermitteln.

<sup>3</sup> Vgl. <http://doc.trolltech.com/4.4/tutorials.html>.

Es ist leicht nachvollziehbar, dass die Vernetzung zwischen dem eigenen Quelltext und den Qt-Quelltext-Dateien relativ ausgeprägt ist und ein Makefile alle diese Verflechtungen und Abhängigkeiten enthalten muss. Qt stellt mit `qmake` ein Werkzeug zur Verfügung, um Makefiles automatisch zu generieren. Dafür wird eine `.pro`-Datei benötigt, die der Entwickler als Konfigurationsdatei anlegt, um sein Projekt zu beschreiben. Darin sind Quelltext- und Header-Dateien, zusätzliche Bibliotheken und Compiler-Flags anzugeben. Die Konfigurationsdatei wird initial im Projektverzeichnis mit folgendem Befehl erzeugt:

```
$ qmake -project
```

Wird dies für das Beispiel `qt.basic.hello` ausgeführt, so entsteht die Datei `qt.basic.hello.pro` mit folgendem Inhalt:

```
#####
# Automatically generated by qmake (2.01a) Mo Apr 28 18:10:39 2008
#####

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
SOURCES += main.cpp
```

Auf einen wiederholten Aufruf von `qmake -project` zu einem späteren Zeitpunkt sollte verzichtet werden, weil damit alle manuell hinzugefügten Inhalte verloren gehen. Weitere Ergänzungen, die der Anwender eintragen könnte, sind:

```
HEADERS += myclass.h # Header-Dateien
FORM = gui.ui # Mit Qt Designer erstellte Oberflächendatei
CONFIG += qt # erlaubt dem Benutzer Einstellungen an
# der Variable QT vorzunehmen
CONFIG += debug # Bewirkt eine Übersetzung im Debug Modus
QT += network # Linkt gegen Modul QtNetwork (für Sockets)
LIBS += -L/usr/local/lib -lmath # Linkt gegen externe Bibliothek libmath
```

Durch eine Ausführung der folgenden Befehle wird zunächst die Erzeugung des eigentlichen Makefiles veranlasst. In einem zweiten Schritt wird dieses dann als Vorlage verwendet, um die Anwendung zu übersetzen:

```
$ qmake
$ make
```

Das Programm sollte nun als ausführbare Datei `qt.basic.hello` vorliegen und durch folgenden Aufruf einen Hello-World-Knopf auf dem Bildschirm darstellen:

```
$ ./qt.basic.hello
```



Abb. D.1. Qt-Fenster mit Hello-World-Knopf (vgl. Beispiel `qt_basic_hello`).

D.1.2 Signals und Slots

Mit dem Prinzip der *Signals und Slots* besitzt Qt eine komfortable und sichere Möglichkeit, um ereignisgesteuert zwischen Objekten zu kommunizieren (siehe Abbildung D.2). Dies ist vielleicht die wichtigste Eigenschaft von Qt und soll deshalb kurz erläutert werden. Signals und Slots stellen eine Alternative zu den üblicherweise verwendeten Rückruffunktionen (*Callback-Funktionen*) dar. Eine Rückruffunktion ist ein Zeiger auf eine Funktion, welcher als Parameter an eine andere Funktion übergeben wird. Dadurch ergibt sich die Möglichkeit, beim Eintreten einer bestimmten Bedingung die Rückruffunktion auszuführen, und diese so bspw. über ein bestimmtes Ereignis in Kenntnis zu setzen.

Ein grundlegendes Problem hierbei ist, dass Rückruffunktionen nicht typsicher sind, da nicht garantiert werden kann, dass ein Aufruf die korrekten Attribute enthält. Weiterhin ist dieses Prinzip nicht wirklich flexibel, da der aufrufenden Funktion alle Rückruffunktionen vorab bekannt sein müssen – es existiert also eine hartcodierte Verbindung. Der Signals-Slots-Mechanismus vermeidet diese Nachteile und ermöglicht typsichere Verbindungen mit loser Kopplung. Der Vollständigkeit halber soll gesagt werden, dass die Verwendung von Signals und Slots mit etwas mehr Rechenaufwand verbunden ist. Die Geschwindigkeitseinbußen sind jedoch in der Praxis nicht relevant.

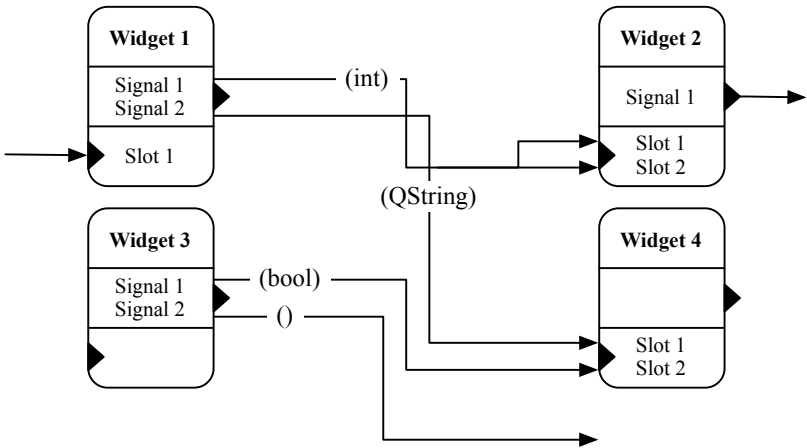


Abb. D.2. Kommunikation zwischen den Qt-Widgets über Signals und Slots.

`QObject` ist die Basisklasse für alle Qt-Objekte, und nur davon abgeleitete Objekte können Signals und Slots besitzen. `QWidget` ist eine Erweiterung von `QObject` und bildet die Basisklasse für alle Objekte, welche auf dem Bildschirm angezeigt werden. Wird wie im folgenden Beispiel von der Klasse `QObject` abgeleitet, so können in der Klassenbeschreibung Signals und Slots unter Verwendung der entsprechenden Schlüsselwörter deklariert werden:

```
#include <QObject>

class Counter : public QObject
{
    Q_OBJECT

public:
    Counter() { m_value = 0; }

    int getVal() const { return m_value; }

public slots:
    void incVal() { m_value++; }
    void decVal() { m_value--; }
    void setVal(int value);

signals:
    void valueChanged(int value);

private:
    int m_value;
};
```

Das Makro `Q_OBJECT` muss in jeder Klasse enthalten sein, die Signals und Slots verwendet. Darin werden zusätzliche Methoden aus dem *Meta-Object-System* von Qt deklariert, die für die Kommunikation notwendig sind. Die Implementierung der Slots erfolgt vergleichbar zu jener anderer Member-Funktionen. Der Befehl `emit <signal_name>(<value>)` löst ein Signal aus, welches an angeschlossene Objekte weitergeleitet wird. Trägt das Signal einen Wert, so wird dieser als Argument übergeben. Eine Implementierung des Slots `setVal(int value)` könnte bspw. wie folgt aussehen:

```
void Counter::setVal(int value) {
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value);
    }
}
```

Die Typsicherheit wird erreicht, indem nur Signals und Slots verbunden werden können, welche die gleiche Signatur besitzen. Da ein Slot zusätzliche Argumente ignorieren kann, ist es möglich, Signals mit größerer Signatur auf Slots mit kleinerer Signatur anzuwenden. Eine Voraussetzung hierfür ist aber, dass die Typen im gemeinsamen Teil übereinstimmen. An einen Slot ohne Attribut kann folglich jedes Signal angeschlossen werden.

Eine Verbindung zwischen Signals und Slots wird über den Befehl `connect()` erstellt. Die Zuordnung erfolgt dabei immer nach demselben Schema

`connect(<from>, SIGNAL(<type>), <to>, SLOT(<type>))` und kann für das obige Beispiel so aussehen:

```
Counter c1, c2;
QObject::connect(&c1, SIGNAL(valueChanged(int)), &c2, SLOT(setVal(int)));
```

Bei dieser Vorgehensweise besitzen die Objekte tatsächlich keinerlei Kenntnis von ihrer Verbindung und sind entsprechend auch unabhängig. An einen Slot können grundsätzlich mehrere Signals angeschlossen werden, und ebenso ist die Verbindung zweier Signals möglich. Das Zielsignal wird dann automatisch mit dem Quellsignal emittiert. In der Regel werden Signals umgehend ausgewertet und sind damit unabhängig von anderen *Event*-Schleifen (GUI-Aktualisierung / Redraw). Die Art der Verbindung kann über `Qt::ConnectionType` festgelegt werden. Über sog. *Queued Connections* werden Signals gesammelt und erst dann an Slots übermittelt, wenn die *Event*-Schleife aufgerufen wird.

Nach einem Übersetzungsprozess mit `Q_OBJECT`-Makro befinden sich im Verzeichnis Dateien der Form `moc_<name>`. Generiert wurden diese Dateien vom *Meta-Object-Compiler (MOC)*, der die zusätzlichen Qt-Erweiterungen einbringt. Er erstellt aus jeder C++-Header-Datei mit `Q_OBJECT`-Makro eine weitere Quelltext-Datei mit Namen `moc_<name>.cpp`, in welcher der *Meta-Object-Code* für diese Klasse enthalten ist. Der MOC wird unter anderem zur Umsetzung des Mechanismus' der Signals und Slots benötigt. Dieser Kompilierungsschritt läuft im Hintergrund ab und ist für den Anwender transparent; die notwendigen Aufrufe werden von `qmake` in das Makefile integriert.

Der Hallo-Welt-Knopf aus dem ersten Beispiel sieht nett aus, erfüllt aber noch keinerlei Funktion ohne Verbindungen zwischen den Widgets. Im Beispiel `qt_basic_signals` wird nun anstelle des Widgets `QPushButton` ein eigenes Widget `MyWidget` definiert. Der Grund für die Erstellung eines eigenen Widgets ist, dass eine Verbindung zwischen `QWidgets` nur innerhalb eines `QWidget` selbst definiert werden kann und nicht in der `main()`-Methode. Abgeleitet von `QWidget` kann dieses Element später als eigenes *Top-Level*-Fenster angezeigt werden. Der folgende Quelltext zeigt die Implementierung der Klasse `MyWidget` (vgl. Datei `qt_basic_signals/main.cpp`):

```
#include <QApplication>
#include <QFont>
#include <QPushButton>
#include <QLCDNumber>
#include <QSlider>
#include <QWidget>

class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
};

MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    setFixedSize(250, 120);
```

```

QPushButton *quit = new QPushButton("Quit", this);
quit->setGeometry(80, 20, 100, 30);

QSlider *slider = new QSlider(Qt::Horizontal, this);
slider->setRange(0, 99);
slider->setValue(0);
slider->setGeometry(20, 75, 100, 30);

QLCDNumber *lcd = new QLCDNumber(this);
lcd->setSegmentStyle(QLCDNumber::Filled);
lcd->setGeometry(130, 75, 100, 30);

connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
}

```

Bei der Betrachtung der Klasse `MyWidget` fällt zunächst der Konstruktor auf: Hier wird standardmäßig ein Parameter `parent` mitgegeben, welcher auf das Vater-Objekt zeigt. In diesem Fall handelt es sich um ein *Top-Level*-Fenster. Es gibt entsprechend keinen Vater und damit bleibt `parent=0`. Im Konstruktor selbst werden drei Elemente erzeugt: Ein Knopf vom Typ `QPushButton`, ein Schieberegler vom Typ `QSlider`, und eine numerische Anzeige vom Typ `QLCDNumber`. Wie schon zuvor, so sind auch hier für jeden Typ die entsprechenden Header-Dateien einzubinden. Diverse geometrische Funktionen setzen das Widget auf eine feste Größe (`setFixedSize(dim_x,dim_y)`) oder positionieren einzelne Elemente und definieren deren Abmessungen (`setGeometry(pos_x,pos_y,dim_x,dim_y)`). Mit den Funktionen `setRange()`, `setValue()` oder `setSegmentStyle()` werden Wertebereiche, Anfangswerte und Stile der einzelnen Elemente gesetzt.

Am Ende des Konstruktors wird der *Quit*-Knopf über den Befehl `connect()` mit der Anwendung verbunden. Im vorliegenden Falle soll das Signal `clicked()` des Knopfes ein Schließen der Anwendung bewirken. Wichtig ist dabei, dass die Variablentypen der Signals und Slots identisch sind, da sonst keine Zuweisung vorgenommen werden kann. Die Auswertung der Verbindungsangaben findet zur Laufzeit statt, sodass erst die Qt-Ausgabe in der Shell auf Probleme hinweist, nicht aber der Compiler. In der zweiten Verbindung wird der Schieberegler an das Anzeigeelement `QLCDNumber` gekoppelt. Sobald `slider` durch eine Wertänderung das Signal `valueChanged()` emittiert, wird der Wert selbst zum Anzeigeobjekt `lcd` übertragen. Das Beispiel zeigt anschaulich den großen Vorteil der Signals und Slots. Widgets mit völlig unterschiedlicher Aktivität, die durch den Anwender ausgelöst werden, können über Signals einfach und typsicher miteinander kommunizieren. Hierzu ist weder Polling, noch die Ereignisübermittlung mit Callback-Funktionen notwendig. Nach einem Aufruf folgender Befehle sollte sich ein Fenster ähnlich wie in Abbildung D.3 zeigen:

```

$ qmake
$ make
$ ./qt_basic_signals

```



**Abb. D.3.** Kommunikation zwischen Widgets unter Verwendung von Signals und Slots im Beispiel `qt_basic_signals`.

### D.1.3 Ein universelles Qt-Makefile

Bisher musste für jeden Übersetzungsdurchlauf der Ablauf `qmake - make` befolgt werden. Um diese Prozedur zu vereinfachen und um leicht neue Quelltext-Dateien hinzufügen zu können, kann das von Qt erzeugte Makefile aber auch hinter einem universellen Makefile versteckt werden. Das Qt-Makefile wird dabei, falls notwendig, von `qmake` neu erzeugt und direkt im Anschluss von `make` zur Übersetzung verwendet. Damit lässt sich der Build-Prozess für eine Anwendung leicht in übergeordnete Makefiles integrieren. Erreicht wird dies mit folgendem universellem Makefile:<sup>4</sup>

```
# This Makefile generates a Qt-Makefile named .Makefile.Qt using qmake and
# builds the application with it. You only need to specify the target and
# have a .pro file with this name.

TARGET = qt_basic_signals

CC = g++
INCPATHS =
LIBPATHS =
LIBS =
FLAGS = -O2 -g
LDFLAGS =

all: .Makefile.Qt
    $(MAKE) -f .Makefile.Qt -j4

.Makefile.Qt: $(TARGET).pro
    qmake -recursive -o .Makefile.Qt

clean: .Makefile.Qt
    make -f .Makefile.Qt clean
    -find . -name '.Makefile.Qt*' -exec rm -f {} \;
    rm $(TARGET)

.PHONY: all clean run
```

Für eine Verwendung in anderen Projekten muss lediglich die Variable `TARGET` definiert werden und eine Projektdatei `.pro` mit gleichem Namen vorhanden

<sup>4</sup> Zu finden im Beispiel `qt_basic_signals` als `Makefile.better`.



sein. Es ist zu beachten, dass ein manueller Aufruf von `qmake` ohne Option `-f .Makefile.Qt` wiederum eine Datei `Makefile` erzeugen und die vorhandene überschreiben würde. Wird das generische Makefile verwendet, so lässt sich das Beispiel durch einen einmaligen Aufruf von `make` übersetzen:

```
$ make clean
$ cp Makefile.better Makefile
$ make
```

## D.2 Oberflächenerstellung mit Qt Designer

### D.2.1 Installation und Grundlagen

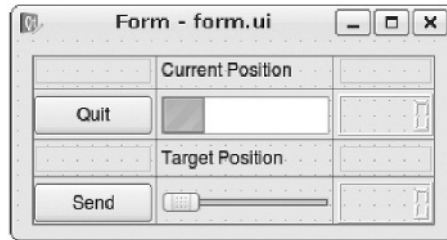
Der *Qt Designer* wurde zu Beginn des Kapitels bereits installiert und wird unter Ubuntu wahlweise über den Menüpunkt *Anwendungen/Entwicklung* oder durch den Aufruf von `designer` in der Kommandozeile gestartet. Die Erstellung einer eigenen Qt-Oberfläche ist denkbar einfach: Nach der Erzeugung eines neuen Formulars vom Typ *Widget*<sup>5</sup> im Menüpunkt *Datei* können auf der gerasterten Oberfläche Elemente aus der Widget-Liste per *Drag & Drop* platziert werden. Im Menü *Bearbeiten* muss dazu der Punkt *Widgets bearbeiten* aktiviert sein. Für verschiedene Aufgaben stehen insgesamt vier Modi zum Editieren zur Verfügung (über den Menüpunkt *Bearbeiten*):

- Im Modus *Widgets bearbeiten* wird das Erscheinungsbild festgelegt, es werden Layouts und Elemente hinzugefügt und die Eigenschaften der Widgets werden spezifiziert.
- Im Modus *Signals und Slots bearbeiten* können die Signals und Slots der Widgets bereits im Qt Designer verbunden werden.
- Der Modus *Buddies bearbeiten* dient dazu, für ein Objekt vom Typ `QLabel` einen zugehörigen *Buddy* zu definieren (bspw. ein `QSpinBox`-Objekt). Über diese Beziehung kann in der Anwendung der Fokus für ein Eingabefeld über die Schnell taste des `QLabels` gesetzt werden. Die Schnell taste wird definiert durch ein Setzen von `&` vor den gewünschten Buchstaben im Namen des Labels, bspw. „N&ummer“ für die Schnell taste ALT-u.
- Im Modus *Tabulatorreihenfolge bearbeiten* wird die Reihenfolge festgelegt, in der die Widgets die Tastaturzuordnung erhalten.

Elementparameter wie Name, Größe, Wertebereich und Farbe können über *Werkzeuge/Eigenschaften* gesetzt werden. Über die Variable `objectName` wird das Element später im Quelltext referenziert, entsprechend ist ein möglichst

<sup>5</sup> Mit Formularen vom Typ *Dialog* ist die Integration in andere Widgets, wie sie später im *Direkten Ansatz* zur Verwendung der Benutzeroberfläche angewandt wird, nicht möglich.

passender Name zu wählen. Mit dem Werkzeug *Signals und Slots* lassen sich die Verbindungen zwischen den Elementen bereits im Qt Designer erstellen. Da aber meist noch weitere Verbindungen benötigt und später im Quelltext codiert werden, ist es oftmals sinnvoller, alle Zuordnungen im Quelltext zusammengefasst durchzuführen.



**Abb. D.4.** Mit Qt Designer erstellte Oberfläche für das Beispiel `udp-qt-server`.

Ein großer Vorteil des Qt Designers zeigt sich bei der Verwendung von *Layouts*. Im Beispiel `qt_udp_server` wurde ein `QGridLayout` noch aufwändig von Hand erstellt. Dieses lässt sich im Qt Designer einfach als Rahmen in der Oberfläche platzieren, um anschließend einzelne Elemente per *Drag&Drop* darin anzuordnen. Die Elemente werden in der Regel der Größe des Layouts angepasst und entsprechend gedehnt. Diese Einstellung kann aber auch durch die Angabe einer Maximalgröße in den Eigenschaften unterbunden werden. Über sog. *Spacer* werden Abstände zwischen Elementen realisiert. Abbildung D.4 zeigt die Oberfläche für das Beispiel `udp-qt-server` als Formular `form.ui`. Die Gitteraufteilung wird durch gepunktete Linien angedeutet.

Hiermit wurde bereits das notwendige Wissen zur Erstellung einer GUI in Form einer `.ui`-Datei vermittelt. Dabei geht es lediglich um die Schnittstelle zum Anwender. Die eigentliche Funktionalität wird später im Programm umgesetzt. Eine Qt-Designer-Oberfläche `sampleform.ui` wird dem Projekt durch folgenden Eintrag in der `.pro`-Datei hinzugefügt:

```
FORMS = sampleform.ui
```

Die Auswertung der Oberflächenbeschreibung geschieht entweder bei der Übersetzung des Projektes (*Compile Time Form Processing*) oder aber erst zur Laufzeit (*Run Time Form Processing*). Die erste Variante ist wesentlich einfacher zu handhaben und auch das Mittel der Wahl für die Beispiele im Buch. Die zweite Möglichkeit wird erst für die Erzeugung dynamischer Oberflächen relevant. Aus der Information in `sampleform.ui` erstellt der *User Interface Compiler (UIC)* eine Header-Datei `ui_sampleform.h`, welche eine Klassenbeschreibung der Oberfläche enthält. Diese kann auch aus mehreren `.ui`-Dateien zusammengesetzt werden. Die Klasse besitzt eine Methode `setupUi()`, über deren Aufruf die Oberfläche mit allen Elementen aufgebaut wird.

Für die Einbindung der Oberfläche in die Anwendung existieren drei Möglichkeiten. Sie werden von der Fa. Trolltech bezeichnet wie folgt:

1. Direkter Ansatz – Erzeugung eines Widgets, in welchem die Benutzeroberfläche angelegt wird:

```
#include "ui_sampleform.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget *widget = new QWidget;
    Ui::FormClass ui;
    ui.setupUi(widget);

    widget->show();
    return app.exec();
}
```

Dieser Ansatz stellt eine schnelle und einfache Möglichkeit dar, um die erstellte Oberfläche anzuzeigen. Der Ansatz ist allerdings auch nur relativ eingeschränkt verwendbar. Beim Errichten innerhalb von `widget` besteht keine Möglichkeit mehr, die Klasse `FormClass` bspw. um eigene Slots oder Signals zu erweitern. Die Oberflächenelemente arbeiten zwar entsprechend der im Qt Designer definierten Verbindungen zusammen, auf einzelne Elemente der Oberfläche kann allerdings nicht zugegriffen werden. Für eine engere Verbindung zwischen Anwendung und Benutzeroberfläche ist daher einer der folgenden beiden Ansätze zu wählen.

2. Ansatz mit Einfachvererbung – Ableiten von `QWidget` und Anlegen der Benutzeroberfläche als private Objektvariable:

```
#include "ui_sampleform.h"

class FormClass : public QWidget
{
    Q_OBJECT

signals:
    // define signals here if needed

public:
    FormClass(QWidget *parent = 0) : QWidget(parent) {
        ui.setupUi(this);
    }

private slots:
    // define slots here if needed

private:
    Ui::FormClass ui;
};
```

Bei diesem Ansatz wird von einem Qt-Widget abgeleitet, und es wird ein Objekt vom Typ `Ui::FormClass` als Objektvariable angelegt. Der Aufruf zur Erstellung der Benutzeroberfläche erfolgt im Konstruktor, weiterhin kann die Klasse um zusätzliche Signals und Slots erweitert werden.

Die Erzeugung der Verbindungen zwischen der Anwendung und den GUI-Elementen ist ebenfalls problemlos möglich. Hierzu muss lediglich der Objektname `ui` als Prefix vorgeschoben werden. Dieser Ansatz eignet sich besonders gut, um mehrere Benutzeroberflächen in einer Klasse zu vereinen.

3. Ansatz mit Mehrfachvererbung – Ableiten von `QWidget` sowie der Klassenbeschreibung der Oberfläche:

```
#include "ui_sampleform.h"

class FormClass : public QWidget, private Ui::FormClass
{
    Q_OBJECT

signals:
    // define signals here if needed

public:
    FormClass(QWidget *parent) : QWidget(parent) {
        setupUi(this);
    }

private slots:
    // define slots here if needed
};
```

Diese dritte Variante hat den großen Vorteil, dass durch Ableiten von `QWidget` und `Ui::FormClass` alle Elemente der Benutzeroberfläche direkt zugänglich sind. Dies bedeutet, dass Verbindungen mit `connect()` auf dem regulären Weg erstellt werden können. Ein Zugriff wird durch die `private`-Deklaration von `Ui::FormClass` auf die eigene Klasse beschränkt. Dieser Ansatz ist grundsätzlich immer vorzuziehen, sofern nicht mehrere Benutzeroberflächen in einem Widget verwaltet werden sollen. In Abschnitt 13.3.2 wird diese Variante für die Umsetzung eines Leitstandes zur Schrittmotorsteuerung verwendet.

### D.2.2 Verwendung der Qt Designer Plugins

Grundsätzlich sind verschiedene Widgets zur Ein- und Ausgabe bereits im Qt Designer enthalten, es fehlen allerdings einige nützliche Elemente wie z. B. LEDs zur einfachen Statusanzeige. Für solche Erweiterungen sind etliche Plugins im Internet frei verfügbar (vgl. z. B. <http://www.qt-apps.org>). Zunächst soll das Plugin `qLED` dieser Seite dem Qt Designer hinzugefügt und dann im folgenden Beispiel `qt_use_plugins` in einer Anwendung verwendet werden. Nach dem Herunterladen und Entpacken von `qLED` stellt man fest, dass zwei Klassen enthalten sind: Das Widget selbst in `qled.h` und die Schnittstellenbeschreibung für den Designer in `qledplugin.h`.<sup>6</sup> Die Projektbeschrei-

<sup>6</sup> Das Plugin ist in `<embedded-linux-dir>/examples/qt/qt_plugins` enthalten.

bung in `qledplugin.pro` zeigt in der ersten Zeile, dass es sich um ein Qt Designer Plugin handelt:

```
CONFIG += designer plugin debug_and_release
```

Das Plugin kann genauso wie jedes andere Qt-Projekt mit den folgenden Befehlen übersetzt werden:

```
$ qmake
$ make
```

Als Ergebnis entsteht die dynamische Bibliothek `libqledplugin.so`. Diese wird nun durch den folgenden Aufruf in das Qt-Verzeichnis unter `/usr/lib/qt4/plugins/designer/` kopiert.

```
$ sudo make install
```

Gleichzeitig werden alle Quelltext-Dateien dieses Beispiels nach `/usr/lib/qt4/examples/designer/qledplugin/` kopiert.<sup>7</sup> Je nach Konfiguration des Plugins in der `.pro`-Datei geschieht das Kopieren der Quelltext-Dateien nicht automatisch, sondern muss per Hand durchgeführt werden. Nach einem Neustart des Qt Designers erscheint das installierte Plugin nun zusammen mit den vorhandenen Widgets in der *WidgetBox* unter der Rubrik *Lab Widgets* – dieser Name wurde in der Schnittstellenbeschreibung in `QLedPlugin::group()` festgelegt.

Wird eine Oberflächenbeschreibung `form.ui` mit dem neuen QLed-Widget erstellt und abgespeichert, so kann diese wie beschrieben über die dritte Methode *Ansatz mit Mehrfachvererbung* in die Anwendung integriert werden. Das Beispiel `qt_use_plugin` zeigt die Vorgehensweise. Nun muss der Pfad zur Header-Datei `qled.h` eingetragen werden, und es muss dem Linker mitgeteilt werden, gegen die dynamische Bibliothek `libqledplugin` zu linken. Dies geschieht durch die folgenden Einträge in der Projektdatei `qt_use_plugin.pro`:

```
INCLUDEPATH += $$[QT_INSTALL_EXAMPLES]/designer/qledplugin/
LIBS += -L$$[QT_INSTALL_PLUGINS]/designer -lqledplugin
```

Die Pfadangaben `$$[...]` werden dabei von `qmake` korrekt aufgelöst. Das Beispiel `qt_use_plugin` sollte samt Formular durch einen alleinigen Aufruf von `make` übersetzen. Nach einem Start von `qt_use_plugin` folgt meist zwangsläufig folgende Fehlermeldung:

```
$ ./qt_use_plugin
./qt_use_plugin: error while loading shared libraries: libqledplugin.so:
cannot open shared object file: No such file or directory
```

Dies resultiert daher, dass die zur Laufzeit benötigte Bibliothek `libqledplugin.so` nicht im Pfad `LD_LIBRARY_PATH` gefunden wurde.

<sup>7</sup> Die Verzeichnisse können je nach Installation variieren, werden von Qt jedoch korrekt gefunden.

Das Hinzufügen des Verzeichnisses `/usr/lib/qt4/plugins/designer/` (mit ggf. abweichendem Namen) schafft Abhilfe:

```
$ export LD_LIBRARY_PATH=/usr/lib/qt4/plugins/designer/:$LD_LIBRARY_PATH
```

Empfehlenswert ist, diese Erweiterung dauerhaft in der Datei `~/.bashrc` einzutragen. Das Beispiel sollte sich nun starten lassen und ein LED-Widget wie in Abbildung D.5 anzeigen.



**Abb. D.5.** Verwendung des Qt Designer Plugins `qLed` im Beispiel `qt.use_plugin`.

### D.2.3 Erstellung der Qt Designer Plugins

Je nach Anwendung kann es wünschenswert sein, eigene Qt Designer Plugins zu erstellen. Da die Qt-Dokumentation hierzu etwas knapp bemessen ist, folgen in diesem Abschnitt zusätzliche Erklärungen am Beispiel des vorgestellten Plugins `qLed`.

Zuvor sei gesagt, dass für die Entwicklung eigener Plugins Erfahrung mit dem *Paint System* von Qt vorhanden sein sollte, da früher oder später der Punkt erreicht wird, an dem grafische Elemente auf dem Bildschirm dargestellt werden sollen. Hierfür bietet das Qt-Tutorial einen guten Einstieg (vgl. URL in Abschnitt D.1.1).

Etwas vereinfacht formuliert handelt es sich bei einem Qt Designer Plugin um ein Qt-Projekt eines bestimmten Typs, welches aus zwei Klassenbeschreibungen besteht. Im Beispiel des Plugins `qledplugin` sind dies die beiden Klassen `QLedPlugin` und `QLed`. Der Qt Designer selbst erwartet eine Beschreibung des Widgets in Form einer von `QDesignerCustomWidgetInterface` abgeleiteten Klasse, in diesem Fall `QLedPlugin`.

Die folgenden Methoden müssen reimplementiert werden, um die für den Qt Designer notwendigen Informationen bereitzustellen:

```

bool isContainer() const;
bool isInitialized() const;
QIcon icon() const;
QString domXml() const;
QString group() const;
QString includeFile() const;
QString name() const;
QString toolTip() const;
QString whatsThis() const;
QWidget *createWidget(QWidget *parent);
void initialize(QDesignerFormEditorInterface *core);

```

Die Implementierung in `qledplugin.cpp` ist nahezu selbsterklärend. So wird in `createWidget()` bspw. auf das eigentliche Widget referenziert:

```

QWidget *QLedPlugin::createWidget(QWidget *parent)
{
    return new QLed(parent);
}

```

Die Angabe der Bilddatei in `icon()` dient der Darstellung eines Symboles in der Liste der verfügbaren Widgets im Qt Designer:

```

QIcon QLedPlugin::icon() const
{
    return QIcon(":/qled.png");
}

```

Eine Standardgröße für die Darstellung im Designer kann als XML-Beschreibung in der Methode `domXml()`, Abschnitt **geometry**, angegeben werden:

```

" <property name=\"geometry\">\n"
" <rect>\n"
" <x>0</x>\n"
" <y>0</y>\n"
" <width>100</width>\n"
" <height>100</height>\n"
" </rect>\n"
" </property>\n"

```

Über das Makro `Q_EXPORT_PLUGIN2(customwidgetplugin, QLedPlugin)` am Ende der Datei `qledplugin.cpp` werden die für den Qt Designer notwendigen Symbole exportiert, um das Plugin darin verwenden zu können. Diese Schnittstellendefinition ist – bis auf die jeweiligen Eigenschaften – für alle Plugins nahezu identisch. Interessanter ist das Widget selbst, dessen Aussehen und Benutzerschnittstelle durch die Klasse `QLed` in `qled.h` beschrieben werden. Die beiden `Q_PROPERTY`-Definitionen in der Klassenbeschreibung dienen dazu, eine generische Schnittstelle für den Zugriff auf `m_value` und `m_color` zu bieten:

```

Q_PROPERTY(bool value READ value WRITE setValue);
Q_PROPERTY(QColor color READ color WRITE setColor);

```

Jede `Q_PROPERTY`-Definition folgt dabei der Syntax:

```
Q_PROPERTY(<type> <name> READ <read_func> [WRITE <write_func>])
```

Nach dieser Definition ist bspw. ein Setzen von `m_color` über folgenden Aufruf möglich, ohne die `set-` und `get-`Methoden der Klasse selbst zu kennen:

```
obj->setProperty("color", Qt::red);
```

Slots für das Setzen der Farbe und des Status' erlauben die Verbindung mit anderen Widgets über `connect()`. In der Klassenmethode `QLed::paintEvent(QPaintEvent *)` wird das Widget auf die Oberfläche gezeichnet, wobei ein Objekt vom Typ `QPainter` mit verschiedenen Zeichenfunktionen für Kreisbögen und Ellipsen diese Aufgabe umsetzt. Der Aufruf von `QWidget::update()` bewirkt, dass zur Ereigniswarteschlange eine an dieses Objekt adressierte Instanz von `QPaintEvent` hinzugefügt und so das Widget beim nächsten Zeichendurchlauf neu erstellt wird.

Das fertige Widget muss über das Makro `QDESIGNER_WIDGET_EXPORT` dem Qt Designer bekannt gemacht werden. Grundsätzlich empfiehlt es sich, ein neues Plugin zunächst als reguläres Widget zu entwickeln und wie folgt zu definieren:

```
#include <QWidget>

class QLed : public QWidget {
    Q_OBJECT
    ...
};
```

Erst wenn dieses getestet ist, sollte es zusammen mit der zugehörigen Plugin-Klasse exportiert werden:

```
#include <QWidget>
#include <QtDesigner/QDesignerExportWidget>

class QDESIGNER_WIDGET_EXPORT QLed : public QWidget
{
    Q_OBJECT
    ...
};
```

Wurden beide Klassen für die Plugin-Beschreibung und das Widget erstellt, so fehlt noch die Projektbeschreibung, bevor das Plugin als dynamische Bibliothek übersetzt werden kann. Bei Betrachtung der Datei `qledplugin.pro` stellt man fest, dass sich auch diese Konfiguration von regulären Qt-Projekten unterscheidet. Durch die folgenden Angaben wird dem System mitgeteilt, dass es sich hierbei um ein Qt Designer Plugin handelt und dass gegen die entsprechende Bibliothek gelinkt werden muss:

```
CONFIG          += designer plugin debug_and_release
TARGET          = $$qtLibraryTarget($$TARGET)
TEMPLATE        = lib
QTDIR_build:DESTDIR = $$QT_BUILD_TREE/plugins/designer
```



Es folgen die Definitionen der Header- und Quelltext-Dateien und die Angabe über sog. Ressourcen:

```
HEADERS    = qled.h qledplugin.h
SOURCES    = qled.cpp qledplugin.cpp
RESOURCES  = qled.qrc
```

Die XML-basierte Datei `qled.qrc` enthält eine Liste der Binärdateien, die in die Bibliothek `libqledplugin.so` integriert werden sollen. Im vorliegenden Falle ist dies das Symbol `qled.png`. Hierdurch wird bewirkt, dass `qled.png` von der Anwendung später durch einen Aufruf von `Q_INIT_RESOURCE(qled)` genutzt werden kann. Eine typische Verwendung dieser Binär-Ressourcen ist das Hinterlegen von Symbolen für Menüzeilen oder Popup-Menüs einer Anwendung. Die fertige Bibliothek muss in ein bestimmtes Verzeichnis installiert werden, um in der Folge vom Qt Designer gefunden zu werden. Weiterhin werden die Header-Dateien für die Verwendung in der eigenen Anwendung benötigt und müssen ebenfalls zentral abgelegt werden.

Folgende Zeilen in `qledplugin.pro` dienen der Festlegung dieser Aktionen und der dafür relevanten Dateien und Verzeichnisse für einen späteren Aufruf von `make install`:

```
target.path    = $$[QT_INSTALL_PLUGINS]/designer
sources.files  = $$SOURCES $$HEADERS *.pro
sources.path   = $$[QT_INSTALL_EXAMPLES]/designer/qledplugin
INSTALLS      += target sources
```

Durch eine Modifikation des Beispiels `qled` sollte es nun einfach möglich sein, auch eigene Qt Designer Plugins zu entwickeln.

# E

---

## Bezugsquellen

Auf den nachfolgenden Seiten haben wir die Bezugsquellen zu den im Buch verwendeten Elektronik- und Mechanikkomponenten zusammengestellt. Tabelle E.1 enthält Computer-Komponenten und Peripherie, die verwendeten I<sup>2</sup>C-Komponenten sind in Tabelle E.2 aufgeführt. Die angegebenen Preise sind als Richtwerte ohne Gewähr zu verstehen. Die Angaben enthalten bereits die gesetzliche Mehrwertsteuer.

Zu einer Übersicht zu alternativen Hardware-Plattformen vgl. auch Anhang B.

Produkt	Hersteller/Bezug	Bestellnummer	Preis (EUR)
MAX233-Wandlerbaustein	http://www.reichelt.de	MAX 233 CPP	4,64
USB-Seriell-Wandler	http://www.reichelt.de	USB2 SERIELL	5,20
Distanzsensor Sharp GP2Y0A02YK	http://www.conrad.de	185364-LN	30,72
Relaiskarte 8-fach seriell	http://www.conrad.de	967720	40,98
BTM222 Bluetooth Modul	http://www.it-wns.de	BTM222	12,99
Navilock GPS-Empfänger USB	http://www.reichelt.de	Navilock 302U	39,50
MicroClient Jr.	http://www.norhtec.com	–	190,00
MicroClient Sr.	http://www.norhtec.com	–	250,00
eBox-2300 (= µC Jr.)	http://store.epatec.net	–	190,00
eBox-2500 (= µC Sr.)	http://store.epatec.net	–	240,00
CF-IDE-Adapter	http://www.pearl.de	PE3213-910	12,90
Hammond-Alu-Profilgehäuse	http://www.farnell.de	1244241	27,30
Logitech Quickcam Express, alte Version	http://www.logitech.com	–	12,00
C-CS-Mount-Adapter	http://www.vd-shop.de	10208	2,98
M2,5-Polyamid-Abstandsbolzen, 12 mm	http://www.ettlinger.de	05.42.120	0,50
USB-A-Lötbuchse, print, stehend	http://www.rmcomputertechnik.de	3574	1,06
2-GB-Intenso-USB-Stick	Media Markt	–	10,00
Bluetooth USB-Dongle Klasse 1	http://www.reichelt.de	DELOCK 61477	12,45
Speedlink UltraPortable USB-Soundkarte	http://www.arlt.de	1120629	14,99
Bluetooth-Tastatur KeySonic 595BT	http://www.reichelt.de	KEYSON ACK595BT	45,95
Bluetooth-Maus Kensington Si670	http://www.reichelt.de	KENS 72271EU	29,95
Longshine LCS-WA5-45 Access Point	http://www.kmelektronik.de	12144	34,91

Tabelle E.1. Komponenten und Bezugsquellen.

Produkt	Hersteller/Bezug	Bestellnummer	Preis (EUR)
C-Control Leistungstreiber	http://www.conrad.de	198280-62	14,31
C-Control Thermometer-Modul	http://www.conrad.de	198298-62	13,28
C-Control Konverter I <sup>2</sup> C zu 1-Wire	http://www.conrad.de	198294-62	22,52
C-Control Stepper-Modul 0,8 A	http://www.conrad.de	198266-62	33,80
I2C-MUX8 HS Bausatz	http://cctools.hs-control.de	1814-2	15,50
I2C-PC-Interface-Bausatz	http://cctools.hs-control.de	1201-1	10,00
I2C-PC-Interface-Fertigmodul	http://cctools.hs-control.de	1201-1	17,00
PC nach I2C-Bus Adapter	http://www.robotikhardware.de	BSPC_I2C	19,80
PCF-AD4 HS I <sup>2</sup> C-Bus AD-Wandler-Bausatz	http://cctools.hs-control.de	1841-1	15,00
PCF-AD4 HS I <sup>2</sup> C-Bus AD-Wandler-Fertigmodul	http://cctools.hs-control.de	1841-1	23,00
Chipkarte 16 kByte (128 kbit)	http://www.reichelt.de	Chipkarte 16kB	1,95
Chipkarten-Kontaktiereinrichtung	http://www.conrad.de	730521-62	9,18
8-Kanal-I <sup>2</sup> C-Umschalter (SMD)	http://www.reichelt.de	PCA 9548 AD	1,80
4-Kanal-I <sup>2</sup> C-Umschalter (SMD)	http://www.reichelt.de	PCA 9544 AD	1,30
8-Bit I <sup>2</sup> CIO-Expander (DIP16)	http://www.reichelt.de	PCF 8574 P	1,30
8-Bit I <sup>2</sup> CIO-Expander (SMD)	http://www.reichelt.de	PCF 8574 T	1,25
8-Bit A/D und D/A-Konverter (DIP16)	http://www.reichelt.de	PCF 8591 P	2,35
8-Bit A/D und D/A-Konverter (SMD)	http://www.reichelt.de	PCF 8591 T	2,20
Integrierter TMC-Schrittmotor-Controller	http://www.reichelt.de	TMC 222 SI	10,90

Tabelle E.2. I<sup>2</sup>C-Komponenten und Bezugsquellen.

# F

---

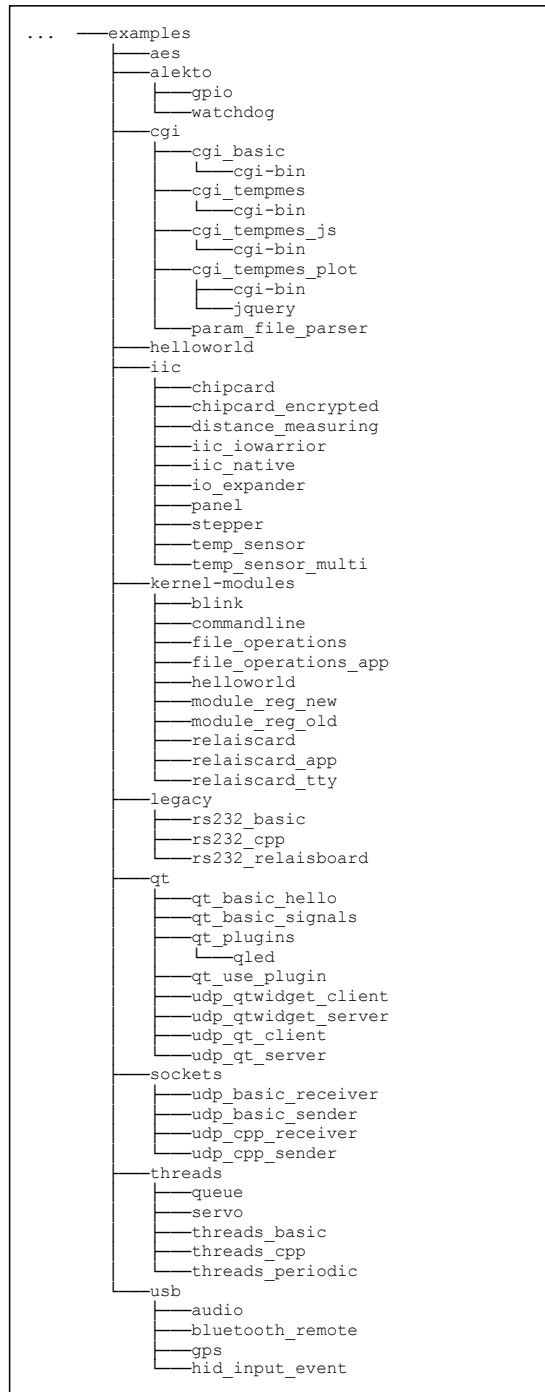
## Verzeichnisbaum

```
embedded-linux-dir
├── datasheets
├── src
│   └── ...
└── examples
    └── ...
```

**Abb. F.1.** Erste Verzeichnisebene.

```
... ──src
      ├── alekto
      │   └── examples
      ├── iic
      ├── iowarrior
      │   ├── iowarrior-2.6
      │   │   └── samples
      │   ├── iowarrior-2.6.20
      │   │   └── samples
      ├── openwrt
      │   ├── helloworld_pck
      │   │   └── helloworld-1.0.0
      │   ├── kmod-iowarrior
      │   │   └── src
      ├── puppy
      ├── video4linux
      └── tools
```

**Abb. F.2.** Zweite Verzeichnisebene: **src**.

Abb. F.3. Zweite Verzeichnisebene: **examples**.

---

## Literaturverzeichnis

- [ALSA 08] ALSA. Projektseite: Advanced Linux Sound Architecture. Online-Quelle. <<http://www.alsa-project.org/>>, 27.01.2008.
- [Ayaz 06] F. Ayaz, D. Koch. Linux konfigurieren und administrieren. Data Becker, Düsseldorf, 2006.
- [Azad 07] P. Azad, T. Gockel, R. Dillmann. Computer Vision – Das Praxisbuch. Elektor-Verlag, Aachen, 2007. <<http://www.praxisbuch.net>>.
- [Beierlein 04] T. Beierlein, O. Hagenbruch. Taschenbuch Mikroprozessortechnik. Hanser Fachbuchverlag, München, 2004.
- [Bluez 08] Bluez. Der offizielle Linux-Bluetooth-Protokoll-Stack. Online-Quelle. <<http://www.bluez.org/>>, 12.10.2008.
- [Bradski 08] G. Bradski, A. Kaehler. OpenCV – Computer Vision with the OpenCV Library. O'Reilly Publishing House, 2008.
- [Brosenne 08] H. Brosenne. Vorlesung zum Thema Betriebssysteme, Universität Göttingen, Institut für Informatik. Online-Quelle. <<http://www.stud.informatik.uni-goettingen.de/os/ws2007/folien>>, 2008.09.19.
- [CC-Tools 07] CC-Tools. Andre Helbig Solartechnik & Energiemanagement. Online-Quelle. <<http://cctools.hs-control.de/>>, 22.01.2008.
- [CiA 08] CiA. Organisation CAN in Automation (CiA). Online-Quelle. <<http://www.can-cia.org/>>, 31.05.2008.
- [Codemercs 08] Codemercs. Webseite der Fa. Code Mercenaries. Online-Quelle. <<http://www.codemercs.com/>>, 31.05.2008.
- [ct 08] ct. Magazin für Computertechnik. Heise-Verlag, Hannover. ISSN: 0724-8679, 2008.10.13.

- [CUPS 08] CUPS. Common UNIX Printing System. Online-Quelle. <<http://www.cups.org/>>, 12.10.2008.
- [Donahoo 08] Jeff Donahoo. Practical C++ Sockets. Online-Quelle. <<http://cs.baylor.edu/~donahoo/practical/CSockets/practical/>>, 31.05.2008.
- [Eclipse 08] Eclipse. Eclipse Projekt - Freie Entwicklungsumgebung. Online-Quelle. <<http://www.eclipse.org/>>, 31.05.2008.
- [Flite 08] Flite. Flite: a small, fast run time synthesis engine. Online-Quelle. <<http://www.speech.cs.cmu.edu/flite/>>, 27.01.2008.
- [GPSD 08] GPSD. GPSD: A GPS service daemon. Online-Quelle. <<http://gpsd.berlios.de/>>, 27.01.2008.
- [Gräfe 05] M. Gräfe. C und Linux. Carl Hanser Verlag, München, 2005.
- [IANA 08] IANA. Internet Assigned Numbers Authority – Vergabe von IP-Adressen. Online-Quelle. <<http://www.iana.org/>>, 31.05.2008.
- [ipkg 08] ipkg. ipkg-Paketverwaltung - Itsy Package Management System. Online-Quelle. <<http://handhelds.org/moin/moin.cgi/Ipkg>>, 31.05.2008.
- [Kingston 08] Kingston. Flash Memory Guide. Online-Quelle. <[http://www.kingston.com/products/pdf\\_files/FlashMemGuide.pdf](http://www.kingston.com/products/pdf_files/FlashMemGuide.pdf)>, 2008.09.30.
- [Linux-Kompendium 08] Linux-Kompendium. Ubuntu / Arbeiten mit dem Terminal. Online-Quelle. <[http://de.wikibooks.org/wiki/Linux-Kompendium:\\_Ubuntu/\\_Arbeiten\\_mit\\_dem\\_Terminal](http://de.wikibooks.org/wiki/Linux-Kompendium:_Ubuntu/_Arbeiten_mit_dem_Terminal)>, 26.03.2008.
- [Liu 73] C. Liu, J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. Journal of the ACM, 20(1):46–61, Jan. 1973. Online erhältlich. <<http://www.cis.upenn.edu/~ishin/cs744/papers/liu-edf-rm.pdf>>, 2008.09.28.
- [Marwedel 07] P. Marwedel. Eingebettete Systeme. Springer-Verlag, Heidelberg, 2007.
- [NIST 08] NIST. National Institute of Standards and Technology (NIST): Advanced Encryption Standard (AES). Online-Quelle. <<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>>, 31.05.2008.
- [NSLU2-Linux 08] NSLU2-Linux. Webseite der NSLU2 Entwickler- und Nutzergemeinde. Online-Quelle. <<http://www.nslu2-linux.org/>>, 02.08.2008.



- [OpenWrt BS Intro 08] OpenWrt BS Intro. Einführung in das OpenWrt-Build-System. Online-Quelle. <<http://forum.openwrt.org/viewtopic.php?pid=31794-p31794>>, 31.05.2008.
- [OpenWrt 08] OpenWrt. OpenWrt Projekt, Linux-Distribution für Embedded-Geräte. Online-Quelle. <<http://www.openwrt.org>>, 2008.05.31.
- [Puppy Linux 08] Puppy Linux. Schlanke Linux-Distribution, läuft fast rein von der RAM-Disk. Online-Quellen. <<http://www.puppylinux.com/>> und <<http://www.puppy-linux.info>>, 26.03.2008.
- [Quade 06] J. Quade, E.K. Kunst. Linux-Treiber entwickeln: Eine systematische Einführung in Gerätetreiber für den Kernel 2.6. Dpunkt Verlag, Heidelberg, 2006.
- [Rayson 08] Rayson. Hersteller serieller Bluetooth-Module. Online-Quelle. <<http://www.rayson.com/product/wireless/BTM-22x.htm>>, 12.10.2008.
- [Reichelt 08] Reichelt. OnlineShop: Reichelt Elektronik. Online-Quelle. <<http://www.reichelt.de/>>, 27.01.2008.
- [SEDeveloper 08] SEDeveloper. Sony Ericsson Developer World. Online-Quelle. <<https://developer.sonyericsson.com/>>, 12.10.2008.
- [I<sup>2</sup>C-Spezifikation 08] I<sup>2</sup>C-Spezifikation. Ausführliche Beschreibung der Fa. Philips. Online-Quelle. <[http://www.nxp.com/acrobat\\_download/literature/9398/39340011.pdf](http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf)>, 07.04.2008.
- [Stroustrup 00] Bjarne Stroustrup. Die C++-Programmiersprache. Deutsche Übersetzung der Special Edition. Addison-Wesley, München, 2000.
- [Subversion 08] Subversion. Versionskontrolle mit Subversion, freies Online-Buch. Online-Quelle. <<http://svnbook.red-bean.com/>>, 31.05.2008.
- [Tanenbaum 02] Andrew S. Tanenbaum. Moderne Betriebssysteme. Pearson Studium, München, 2. überarbeitete Auflage, 2002.
- [TIS 08] TIS. Homepage der Fa. The Imaging Source GmbH in 28215 Bremen. Produktspektrum und Datenblätter. Online-Quelle. <<http://www.theimagingsource.com>>, 2008.10.22.
- [Trinamic 08] Trinamic. Website der Fa. Trinamic. Online-Quelle. <<http://www.trinamic.com>>, 31.05.2008.
- [Vision Systems 08] Vision Systems. Internetauftritt der Fa. Vision Systems GmbH. Online-Quelle. <<http://www.visionsystems.de/>>, 31.05.2008.

- [WhichSBC 08] WhichSBC. Single Board Computer Resource Centre. Online-Quelle. <<http://www.whichsbc.com>>, 31.08.2008.
- [Wikipedia 08] Wikipedia. Die Freie Enzyklopädie (deutsche Version). Online-Quelle. <<http://de.wikipedia.org/wiki/Hauptseite>>, 2007.12.27.
- [Wörn 05] H. Wörn, U. Brinkschulte. Echtzeitsysteme. Springer-Verlag, Heidelberg, 2005.
- [X-Wrt 08] X-Wrt. X-Wrt Projekt, Weboberfläche für OpenWrt. Online-Quelle. <<http://x-wrt.org>>, 2008.05.31.
- [Yaghmour 08] K. Yaghmour. Building Embedded Linux Systems (2. Ed.). O'Reilly Publishing House, Cambridge / USA, 2008. Anm.: Erst in dieser zweiten Auflage wird RT Linux angesprochen.
- [Zahn 06] Markus Zahn. Unix-Netzwerkprogrammierung mit Threads, Sockets und SSL. Springer-Verlag, Heidelberg, 2006.

---

# Sachverzeichnis

1-Wire-Bus .....	155	Bluez-Utils .....	211
8051-Technologie .....	19	Bondout-Chip .....	34
A		Build-Essential .....	80, 91
A/D-Wandler .....	179	C	
Access Point .....	209	CAN .....	153
Advanced Packaging Tool.....	384	Centronics.....	124
AES-Verschlüsselung .....	194	CGI.....	333
Alarmanlage.....	358	Chipkarte .....	190
Alekto .....	51, 87	Chipsätze.....	24
ALSA-Bibliothek .....	204	Clock Stretching .....	137
APT .....	384	Common Gateway Interface .....	333
apt-get .....	385	Communities .....	365
Architekturen.....	18, 75	Controller .....	18
ARM .....	75, 87	Controller Area Network ....	<i>siehe</i> CAN
Assembler .....	19	Cronjobs .....	391
AT-Befehle .....	217	Cross Compiler.....	33, 65, 92
AT24Cxx.....	191	CUPS .....	81
Atheros.....	46	D	
B		D/A-Wandler .....	179
Berkeley Sockets.....	300	Data Structure Alignment .....	316
Betriebssysteme.....	27, 32	Datagramm .....	300
Bewegungserkennung .....	357	Dateiformate .....	227
Bezugsquellen .....	435	Dateioperationen .....	249
Big Endian .....	311	Dateisysteme .....	235
Blacklist.....	344	Deadlock .....	262
Bluetooth .....	210	Debian-Installation	
Dienste, 215		Alekto, 88, 98	
Fernbedienung, 219		NSLU2, 76	
Pairing, 214		Debugging.....	33
Profile, 212		On-Chip, 34	
Serielle Kommunikation, 217		Diagnosemodus.....	393

DIN 44300.....25  
 Domotik ..... 367  
 DotPup ..... 104  
 Druckerschnittstelle ..... *siehe* Parallele  
 Druckerserver.....81  
 DS1631.....177  
 DSP.....17

## E

eBox-2300 ..... 47  
 Echtzeit  
   feste, 26  
   harte, 26, 293  
   weiche, 26, 293  
 Echtzeit-Thread ..... 276  
 Echtzeitbetriebssystem ..... 28  
 Echtzeiterweiterung ..... 290  
 Echtzeitfähigkeit.....25  
 EEPROM-Speicher ..... 190  
 Eingebettete Systeme ..... 17  
 Einplatinencomputer ..... 396  
 Embedded Systems ..... 17  
 Emulation ..... 34  
 Emulator ..... 36  
 Euro-Format ..... 21  
 Event-Schnittstelle ..... 221

## F

Flash-Speicher ..... 22  
 Flash-Speicher-Backup.....84  
 Formatierung.....227  
 Formfaktor ..... 21

## G

Gerätedatei.....247, 387  
 Gerätemanagement ..... 236  
 Gerätenummer ..... 243  
 Gerätetreiber ..... 231  
   Registrierung, 245  
   Verwendung, 255  
 GNU General Public License ..... 238  
 GPIO ..... 127  
 GPS-Empfänger ..... 222  
 GPSD ..... 223  
 GSPCA ..... 342

## H

Hardware Abstraction Layer (HAL) 233  
 Hardware-Plattformen ..... 41

Hardware-Zugriff ..... 257  
 HD44780 ..... 171  
 High Resolution Timer ..... 290  
 Home-Verzeichnis.....228  
 Host Byte Order ..... 311

## I

I<sup>2</sup>C ..... 95, 133

  A/D- und D/A-Wandler, 179  
 Adressierung, 138  
 Anbindung (IO-Warrior), 149  
 Anbindung (nativ), 147  
 Arbitrierung, 138  
 Bibliothek, 163  
 Chipkarte, 190  
 Clock Stretching, 137  
 Display, 170  
 I/O-Erweiterung, 167  
 Komponenten, 161, 435  
 Leitungstreiber, 143  
 Multi-Master-Betrieb, 138  
 Multiplexer, 141, 199  
 Puffer, 145  
 Repeated-Start-Condition, 140  
 Schrittmotorsteuerung, 184  
 Spezifikation, 135  
 Steckverbindung, 146  
 Steuersignale, 136  
 Tastatur, 169  
 Thermostat, 177  
 Verlängerung, 142

ICE ..... 34

IEEE 1284 ..... 124

ifconfig ..... 380

In-Circuit-Emulation ..... 34

Industrielle Kompaktsysteme ..... 396

Input-Subsystem ..... 221

Intel Atom ..... 52

Intelligente Kamera ..... 355

Inter-Integrated Circuit ..... *siehe* I<sup>2</sup>C

Interrupt ..... 19

IO-Ports ..... 257

IO-Speicher ..... 257

IO-Warrior ..... 71, 129

  I<sup>2</sup>C-Bibliothek, 150

  Installation, 130

ioctl() ..... 94, 254

ipkg ..... 63

- ISO/OSI-Modell ..... 297
- ITX-Format ..... 21
- IVT ..... 352, 356, 362
- J
- JavaScript ..... 338
- jQuery-Framework ..... 339
- JTAG-Interface ..... 34
- K
- Kühlung ..... 21
- Kamera
  - Firewire, 361
  - Hardware-Trigger, 360
  - IEEE 1394, 361
  - USB, 361
- Kernel
  - Übersetzen, 97
  - Aufbau, 234
  - Quellen, 97
- Kernel-Space ..... 233
- Kernel-Timer ..... 259
- Kerneldebugger ..... 241
- Kernelmodul ..... 231
  - Übersetzung, 239
  - Argumente übergeben, 242
  - Aufbau, 237
  - Debugging, 239
  - Programmierung, 237
- Komponenten ..... 435
- L
- Latenz ..... 290
- LC-Display ..... 170
- leanXcam ..... 368
- Legacy-Schnittstellen ..... 111
- Linux-Befehle ..... 373
- Linux-Konsole ..... 373
- Little Endian ..... 311
- Local Interconnect Network (LIN) .. 154
- Loopback-Schnittstelle ..... 380
- LPT ..... 124
- lsusb ..... 341
- M
- Majornummer ..... 243
- Matrix-Tastatur ..... 169
- MAX233 ..... 55
- Mechanik ..... 21
- Meta-Object-Compiler (MOC) ..... 422
- MicroClient ..... 47, 101
- Microwire ..... *siehe* SPI
- Mikrocontroller ..... 18
- Mini-PCI ..... 46
- Minornummer ..... 243
- MIPSEL ..... 58
- mknod ..... 387
- MLC-Technologie ..... 23
- MPlayer ..... 342
- Multitasking ..... 263
- Multithreading ..... 263
- Mutex ..... 267, 277
- N
- nano-Editor ..... 379
- NAS ..... 42, 75, 395
- Network Byte Order ..... 311
- Network-Attached-Storage... *siehe* NAS
- Netzwerk
  - Byte Order, 311
  - Datenübertragung, 297
  - Eigene Protokolle, 313
  - Kommunikation mit Qt4, 321
  - Kommunikationsablauf, 301
  - Protokollfamilie, 310
  - Schichten, 297
  - Schnittstelle, 295
- Netzwerkeinstellungen ..... 380
- Netzwerkmanagement ..... 236
- NMEA-Protokoll ..... 222
- NSLU2 ..... 42, 75, 394
- NTP ..... 79
- NUXI-Problem ..... 311
- O
- ObexFTP ..... 216
- Objektiv
  - Anschlussformat, 359
  - Auflagemaß, 359
  - C-Mount, 359
- Open Source Automation Dev. Lab .. 367
- OpenCV ..... 362
  - Lizenz, 362
- OpenMoko ..... 367
- OpenRemote ..... 367
- OpenRISC ..... 51, 87
- OpenWrt ..... 57
- OSADL ..... 367

OTP-Speicher ..... 189

## P

Padding ..... 316  
 Parallele Schnittstelle ..... 124, 258, 286  
 Parität ..... 113  
 Partitionierung ..... 226  
 PATH ..... 386  
 PCA9548 ..... 199  
 PCF8574 ..... 167  
 PCF8591 ..... 179  
 Pegelanpassung ..... 55  
 Periodischer Thread ..... 282  
 PetGet ..... 104  
 Posix-Threads ..... 269  
 Practical Sockets ..... 312  
 proc-Erweiterungen ..... 93  
 Prozess-Scheduler 29, 234, 263, 265, 289  
 Prozess-Scheduling ..... 29  
 Prozessmanagement ..... 234  
 Prozessoren ..... 18  
 Puppy Linux ..... 101

## Q

Qt Designer ..... 320, 328  
 Qt Designer Plugin... 320, 328, 428, 430  
 Qt Widget ..... 421  
 Qt-Framework ..... 320  
 Query String ..... 334

## R

Race Condition ..... 267  
 Real-time Linux Foundation ..... 366  
 Realzeituhr ..... 43  
 Ressourcenverteilung ..... 266  
 Root-Rechte ..... 390  
 Router ..... 395  
 RS-232  
   Übertragung, 113  
   Kabellängen, 114  
   Programmierung, 116  
   Relaiskarte, 121  
   Steckerbelegung, 115  
   Terminalattribute, 118  
   Wandler, 112  
 RS-485 ..... 153

## S

Schedule ..... 31

Scheduler ..... *siehe* Prozess-Scheduler  
 Scheduling ..... 29

  dynamisches, 32  
   First Come First Served, 30  
   non-preemptive, 29  
   preemptive, 29  
   Rate-Monotonic, 30  
   RMS, 30  
   Round-Robin, 30  
   Shortest Job First, 30  
   statisches, 31

Schnittstellen ..... 23

Schrittmotoransteuerung ..... 184, 327

Secure Shell ..... *siehe* SSH

Serial Peripheral Interface .... *siehe* SPI

Serialisierung ..... 315

Serielle Bussysteme ..... 152

  Übersicht, 158

Serielle Konsole ..... 85

Serielle Relaiskarte ..... 260

Serielle Schnittstelle ..... *siehe* RS-232

Servo-Ansteuerung ..... 284

Servo-Steuersignal ..... 286

Signals und Slots ..... 420

Simulator ..... 35

SLC-Technologie ..... 23

Smart Camera ..... 355, 397

Smart Phone ..... 396

Socket ..... 295

Socket-Debugging ..... 310

Software-Entwicklung ..... 33

Speichermedien ..... 22

Speicherverwaltung ..... 235

Spezialdatei ..... *siehe* Gerätedatei

SPI ..... 156

Spinlock ..... 262

Sprachausgabe ..... 206

SSH ..... 64, 382

Stromversorgung ..... 24

stty ..... 120

sudo ..... 390

System Call Interface (SCI) ..... 233

Systemarchitektur ..... 232

Systemressourcen ..... 257

## T

Task ..... 234, 264

TCP ..... 299

- TCP/IP-Referenzmodell ..... 297
- Temperaturmessung ..... 177, 336
- Thread ..... 234, 264, 275
- Thread-Priorität ..... 276
- TMC222 ..... 184
- Toolchain ..... 92
- U
- UART ..... 113
- UDP ..... 297, 299
- Umgebungsvariablen ..... 386
- Unicap-Treiber ..... 361
- Universal Serial Bus ..... *siehe* USB
- Upslug2 ..... 77
- USB ..... 156, 203
- USB-Adapter ..... 203
  - Audio, 204
  - Bluetooth, 210
  - GPS, 222
  - Speicher-Stick, 226
  - WLAN, 206
- User Interface Compiler (UIC) ..... 426
- User-Space ..... 233
- V
- V4L ..... 345, 346
- V4L2 ..... 345, 351
- Verklemmungen ..... 268
- VESA-Format ..... 21
- vi-Editor ..... 377
- Video for Linux ..... 345
- videodog ..... 343, 345
- W
- Warteschlange ..... 280
- Watchdog ..... 24, 96
- Wear Leveling ..... 23
- Webcam ..... 341, 345, 355
  - Bildqualität, 344
  - Parameter, 344, 351
- Webprogrammierung ..... 333
- Webserver ..... 333
- WL-500 ..... 44, 57, 393
- WLAN-Treiber ..... 206
- WRT54 ..... 393
- Z
- Zeichenorientierte Gerätetreiber .... 243
- Zeitmessung ..... 290
- Zeitverhalten ..... 25
- Zugriffsmodus ..... *siehe* Zugriffsrechte
- Zugriffsrechte ..... 247, 388
- Zustandsvariablen ..... 269, 279